Relations and Functions 1

Today:

- ☐ Relations
- ☐ Representing relations
- ☐ Equivalence relations and equivalence classes
- ☐ Integers mod n
- ☐ Functions
- ☐ Computing functions you don't understand

I started this lecture content last Thursday (4R, January 27), but didn't get far. So I have rolled those notes into today's.

1 Relations

Relations play an important role in discrete math. They capture situations in which there is a universe of "things" and two of these things, a and b, might or might not be related to one another in some way. A building might or might not be on a block; two cities might or might not have a connecting flight; one person might or might not know a second; a person may or may not be the parent of another. That sort of thing. Beyond this, we use relations to define functions, which we use pretty much constantly in mathematics, and are central to how mathematicians think.

Definition: With A and B nonempty sets, a (binary) *relation* R is a subset of $A \times B$. That is, any $R \subseteq A \times B$ is a relation.

When $(a, b) \in R$ we say that *a is related to b* under R. Note the ordered pair: just because a is related to b doesn't mean that b is related to a.

Usually we prefer to write things in infix notation, writing $x \; R \; y$ instead of $(x, y) \in R$. And often we use symbols, rather than letters, for relations: for example, $\sim$ or $<$. Then we might write $x \sim y$ in place of $(x, y) \in \sim$, or $x < y$ in place of $(x, y) \in <$, which looks pretty weird.

Here are some common relations you know from arithmetic, for comparing numbers, where the underlying sets $A = B$ are the sets of natural numbers, integers, or reals: $=$, $<$, $\leq$, $>$, $\geq$.

Another important relation for integers is the *divides* relation, $|$. We say that $d \mid m$, read *d divides m*, to mean that there exists a number i such that $id = m$.

What about our friends succ (the successor function), $+$ (addition), and $*$ (multiplication)? No, these are function symbols, not relations.

In set theory we had a single relation symbol, $\in$. But we used it to define other.

What about $\emptyset$? No, that's a constant symbol.

Often when dealing with a relation $R \subseteq A \times B$ we have $A = B$. That is the case for all of the following examples.

1. The integers $\mathbb{Z}$ along with $\leq$. **Not an equivalence relation.**
2. The set of strings over some alphabet, Σ^*, and $x \leq y$ if x a *substring* of y. We say that x is a substring of y if $y = uxv$ for some strings u and vv. **Not an equivalence relation.**
3. The set of lines in the plane, with $x \sim y$ if they are parallel Is a line parallel to itself? Let's say that it is. **An equivalence relation.**
4. The set of strings over some alphabet, and x and y are related if they have the same length. **An equivalence relation.**
5. The set of integers, and $a \; R_d \; b$ if $d \mid (a - b)$ **An equivalence relation.**
6. The set of real numbers, and $a \sim b$ if $\lfloor a \rfloor = \lfloor b \rfloor$. By $\lfloor x \rfloor$ we mean the largest integer that is less than or equal to x. **An equivalence relation.**
7. The set of cities in the U.S. and $a \sim b$ if it is possible to get from a to b on a Greyhound bus. **An equivalence relation.**
8. Relations on a finite set, say $\{0, 1, 2, 3\}$, where $\sim = \{(0, 1), (1, 2), (2, 3), (3, 0)\}$. **Not an equivalence relation.**

2 Representing Relations

If $R \subseteq X \times Y$ then it is natural to represent the relation with a *bipartite directed graph*: the vertices (points) are the "disjoint union" of X and Y, and there's a directed edge (an arc) from $a \in X$ to $b \in Y$ if $a \; R \; b$. Give some examples.

For the case of $R \subseteq X \times X$ the natural way represent the relation is with a *directed graph* where the vertices are X there's a directed edge (an arc) from $a \in X$ to $b \in X$ if $a \; R \; b$. Give some examples.

3 Equivalence Relations and Equivalence Classes

The classical approach of presenting equivalence relations first and later speaking of partitions—if at all!—leaves the former inadequately motivated. A better way to get at it is to start with equivalence classes.

We have some set of things, and we want to regard some of those things as equivalent to other things. "Things that have the same remainder when you divide by 3." "Things that appear to be the same kind of object (fire hydrant, bicycle, traffic signal)." "Elements that have 7 electrons in the their outer shell" (I'm probably showing my age by speaking of electrons orbiting in shells.) "Home prices of 0–500K, 501K–999K, 1M-10M." So we assert when two things are *equivalent*, and then we break our universe into those classes.

What would a relation need so that it would serve to partition our universe X? The following suffice. And are necessary, too.

Definition: A relation $R \subseteq X \times X$ is an *equivalence relation* if it has the following three properties:

- **Reflexive property**: $a\ R\ a$ for all $a \in X$.
- **Symmetric property**: $a\ R\ b \to b\ R\ a$, for all $a, b \in X$.
- **Transitive property**: $a\ R\ b\ \wedge\ b\ R\ c \to a\ R\ c$ for all $a, b, c \in X$.

Explain what each property means, and give examples. Go back and mark equivalence relations from the prior list.

What does being an equivalence relation mean in terms of the directed graph? There's a self loop at every vertex (so maybe one wouldn't bother to draw those arcs); arcs come in matched pairs (whence they can lose their arrows and be thought of as undirected); and we must "close" components to make a collection of disjoint "cliques."

If R is an equivalence relation over $A \times A$ then $[a]$ denotes the set of all elements related to a: that is, $[a] = \{x\colon\ a\ R\ x\}$. Which is of course the same as $[a] = \{x : x\ R\ a\}$. You can think of $[a]$ as the set of everything that's related to a.

We call $[a]$ the *equivalence class*, or the *block*, containing a.

Definition: The set of all equivalence classes of A with respect to a relation R is denoted A/R. It is read as *the quotient set of A by R*, or simply A mod R. One speaks of "modding out" by R.

I claim that *every equivalence relation on a set partitions it into its blocks.* Which is really why equivalence relations matter.

What does the mean?! We must explain what it means to partition a set A:

Definition: Let A be a set. Let I be a set (which is simply used to *index*, or name, the blocks). We say that $\{A_i : i \in I\}$ is a *partition* of A if (0) each A_i is a nonempty set, and (1) their union is all of A, $A = \bigcup_i A_i$, but (2) their pairwise intersection is empty, $A_i \cap A_j = \emptyset$ for all $i \neq j$.

This pretty much corresponds to the English-language meaning of the word *partition*. We have broken our set into non-overlapping chunks—we partitioned it up.

Proposition: Let R be an equivalence relation on a set A. Then the blocks of R comprise a partition of A.

Proof: Every element x of A is somewhere in the partition—namely, $x \in [x]$. Thus the block cover all of A—meaning that their union is all of A. Now suppose that $[x]$ and $[y]$ intersect. I need to argue that they are identical. So suppose there exists some point a that is in both $[x]$ and $[y]$: that is, $a \in [x]$ and $a \in [y]$. I must show that $[x] = [y]$, for which it is enough, by symmetry, to show that any $b \in [x]$ is in $[y]$. But $b \in [x]$ means $b\ R\ x$, we know $x\ R\ a$, whence $b\ R\ a$. We also know that $a\ R\ y$, and so $b\ R\ y$, meaning that $b \in [y]$. $\diamondsuit$

In fact, the relation between equivalence relations and partitions goes both ways: Given a partition $\{A_i : i \in I\}$ of a set A, you can define a relation R by asserting that $x\ R\ y$ iff x and y are in the same block of the partition: there exists and i such that $x \in A_i$ and $y \in A_i$. Then R is an equivalence relation.

We can represent equivalence classes and partitions in pictures. The points in the set A are grouped into blocks. Two points are in the same block if they are related to one another under the equivalence relation. Draw some pictures.

Now go back to the prior examples and identify the blocks in each case. For example, suppose strings x and y are equivalent under R if they have the same length. Then the blocks are $[\varepsilon], [a], [aa], \ldots$ where a is an arbitrary character from the alphabet. Here, we are using a nice canonical name for each block. It's good to choose such canonical names.

An important example in formal-language theory: let L be a language and define from it the relation R_L by saying that xR_Ly if for all z, $xz \in L$ iff $yz \in L$. It's a fairly deep and interesting fact that a language is regular exactly when the relation I've just described has a finite number of blocks. This is called the Myhill-Nerode theorem.

Let's do another exercise: figure out the blocks if two strings x and y are related, $x \sim y$, if both have the same length.

And: figure out the blocks if two strings x and y are related if they share the same first two characters. They must each have length at least two.

4 Integers Mod n

Let's go back to that relation on integers that says that, for any fixed $n > 0$, says that $a \ \equiv_n \ b$ if $n \mid (a - b)$. This is the *congruent-mod-n* relation. So $5 \equiv_{10} 25$. Or $5 \equiv 25$ if it's understood that you're thinking of everything mod 10.

Then draw equivalence classes for this $\equiv_n$ relation, partitioning the universe of integers into n classes. You can either think that a and b belong in the same class (set in the partition) if their difference is divisible by n. Or you can think that a and b belong in the same class if the remainder you get when you divide a by 10 is the same value as the remainder you get when you divide b by 10. Try to figure out why these are the same thing.

This example is an important one in mathematics. It's the *ring of integers modulo n.* We call it a ring because it's easy to define two operations on it, addition, and multiplication, and the two operations satisfy required, familiar properties. If you don't care about the multiplication, you can call it the *group* of integers modulo n.

There are several different ways you can understand this thing, this ring of integers modulo n.

1. The way we just described it—as the equivalence classes you get by starting with the integers and modding out with the relation $\equiv_n$ that says $a \equiv_n b$ if $n \mid (a - b)$. Now $5 \equiv_{10} 25$. It's not that 5 and 25 are the they're not in this viewpoint—but they're equivalent with respect to the $\equiv_n$ way of defining equivalence.

2. Alternatively, think of $\mathbb{Z}_n$ as a set of points named $0, 1, \ldots, n-1$ arrayed clockwise in a circle. The point $a + b$ is determined by starting at a and moving clockwise b steps.

   It might be convenient to regard other points in $\mathbb{Z}$ as synonyms for points in $\mathbb{Z}_n$. For example, if $n = 10$ than the point canonically named 2 might also be called 12, 22, or -8. But these are just different names for the same point. Better to use the canonical name. In the world $\mathbb{Z}_{10}$, what is $7 + 7$? It is 4. $7 + 7 = 4$. *Not* $7 + 7$ is in some way being judged equivalent to 4, but that they are equal.

   It's is the way that I myself would usually think of $\mathbb{Z}_n$.

   As for notation, if you want to work in this realm you can just say: "now we are working in $\mathbb{Z}_{10}$, and suddenly $8^2 = 4$, and $6 \cdot 7 = 2$, and so on. As a reminder that we're in this world, you can write "mod10", in parenthesis, at the end of an asserted equality. For example,

   $$6 \cdot 7 = 2 \pmod{10}$$

3. The way people are taught to think about this in programming languages is to think of mod as an operators. It takes in two integers, x and n, with the second one positive. It returns the number $x \bmod n \in \{0, 1, \ldots n-1\}$ that is the remainder you get if you divide x by n.

We'll come back to this when we study the integers. But a theorem there says that for any $n \geq 1$ and any integer x, there exists integers k and d such that $x = kn + d$, where $d \in \{0, \ldots, n-1\}$. It's called the division theorem. You learned how to find k and d in grade school. At least when x is positive.

The value of $x \bmod n$ is the d that is guaranteed by the division algorithm.

A common notation for mod in programming languages is the infix operator `%`. So one might write `36%10` (which is 6). What is `10%3`? It's 1. What about `-10%3`? Get clear why it's 2.

5 Functions

A crucial kind of relation is a *function.*

Definition. A **function** is $f \subseteq A \times B$ is a relation in which for every $a \in A$ is related to exactly one $b \in B$.

We can write the condition in first-order logic:

$$(\forall a \in A)(\exists b \in B)((a,b) \in f) \wedge (\forall a \in A)(\forall b \in B)(\forall b' \in B)((a,b) \in f \wedge (a,b') \in f \to b = b').$$

When f is a function, we write $b = f(a)$ to mean that $(a,b) \in f$.

Another word for a function is a *map.* If $f(x) = y$ we might say that x *maps to* y. We could also say that y is the *image* under f of x. And we could say that x is a *preimage* under f of y.

Notation. When $f \subseteq A \times B$, we call A the *domain* of f. We call B the *codomain* or **target** of f. We usually write $f(x) = y$, and we write $f \colon A \to B$ to indicate f's domain and range.

What does an arrow-diagram look like for a function? Draw some pictures.

Our functions are **total**, meaning that they are defined on every single point in the domain. Sometimes computer scientists like to consider *partial* functions, which have "holes" in their domains—points where they're not *well-defined.* But we won't do that—and those aren't functions you get, they would be *partial* function.

Sometimes when you're trying to define a function it's not clear what "output" you intend for $f(x)$. Maybe it's not clear that $f(x)$ has any value—or maybe it looks like you might be giving multiple values to $f(x)$. We would say that your function is apparently not well-defined.

Example: You try defining a function $f \colon \mathbb{R} \to \mathbb{Z}$ by saying "for all x, let $R(x)$ be the integer whose value is closest to x." The function is not well-defined because it's not clear what is $R(0.5)$, for example. You would need to correct your definition by saying, for example, "for all x, let $R(x)$ be the integer whose value is closest to x. If there are two integers equally close, chose the one that is *larger*." Or we could have said *smaller*. Or we could have said: *closer to 0*. These would give you three different round-to-integer functions. But you can't just say nothing.

You might also have an ill-defined function because you weren't clear about the domain. If you tell me to let inc be the add-1 function, you haven't specified the domain you have in mind. Is it $\mathbb{N}$? $\mathbb{Z}$? Or maybe $\mathbb{Z}_{2^{32}}$ (which is the domain and the interpretation of incrementing when applied to unsigned 32-bit values in a language like C).

You might have have an ill-defined function because you weren't clear on the target of the function. This is usually less of a problem, for it is routinely a matter of choice. The target *has* of a function f with domain A has to include all of the *range* of f, which is defined as $\{f(x) \colon x \in A\}$. But often it is convenient to think of the target as larger than the range, or potentially larger than the range.

We've been dealing with functions all along. An operator like AND was a function $\wedge \colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$. A function like integer addition is a map $+ \colon \mathbb{Z}^2 \to \mathbb{Z}$.

Definition. Let $f \colon A \to B$ be a function. If the range of f is exactly the target of f, $B = \{f(a) \colon a \in A\}$, then we say that f is **onto**, or **surjective**.

Let's make up some more functions. How about the function that maps everyone to their birthday, $b \colon P \to [1..12] \times [1..31]$. So $b(\mathsf{phil}) = (7,31)$, $b(\mathsf{son}) = (5,8)$. Is b onto? Not the way we described it, because nobody has a birthday of $(2,31)$. Could we "clean up" the target to make it only have the 366 "valid" days? Sure.

How about the function that squares a number? Again, specify the domain to get a well-defined function. If we decide that that domain is $\mathbb{Z}_{10}$, say, then squaring is a function $s \colon \mathbb{Z}_{10} \to \mathbb{Z}_{10}$. What do the different points map to? Well, $0 \mapsto 0$, $1 \mapsto 1$, $2 \mapsto 4$, $3 \mapsto 9$, $4 \mapsto 6$, $5 \mapsto 5$, $6 \mapsto 6$, $7 \mapsto 9$, $8 \mapsto 4$, and $9 \mapsto 1$. Notice that I used a different kind of arrow to show the *image* of a point under f. It's $\to$ for indicating the domain and target of a function—$f \colon A \to B$, but it's this $\mapsto$ arrow to tell me what some specific point mapped to, $x \mapsto y$, with the function itself understood.

Can you use this to define a function. Sort of. If I tell you to map $x \mapsto 2x$ in the reals, this is the same as saying $f(x) = 2x$, but it has the advantage, or the disadvantage, of not having to *name* the function. On the other hand, don't say $x \to 2x$. That looks like garbage.

I see lots of "ad hoc" notation when it comes to function. Don't. If you're writing $f(x = a) : b$ or whatever, don't expect any credit at all. To be understood in English you need to speak in fairly grammatical sentences. To be understood in math you need to write or speak fairly grammatical ways, too.

I've told you one important property of a function—whether or not it's surjective (onto). You can express a function $f \colon A \to B$ as being surjective as the condition:

$$(\forall b \in B)(\exists a \in A)(f(a) = b).$$

A second key property of a function $f\colon A \to B$ is whether or not it is *injective.*

An injective function is one without *collisions*, which are distinct points $a, a' \in A$ such that $f(a) = f(a')$. In directed graph that represents a function, it's a target value y that has two *preimages*—points that map to it. Here's a logical definition for when a function $f\colon A \to B$ is injective:

$$(\forall a \in A)(\forall a' \in A)(f(a) = f(a') \to a = a')$$

Another word for a function being injective is it being *one-to-one* Let's memorialize all this:

Definition. A function $f\colon A \to B$ is *injective* (or *one-too-one*) if $f(a) = f(a')$ implies $a = a'$.

Here are some more examples. Let $f\colon \mathbb{N} \to \mathbb{N}$ be defined by $f(x) = x^2$. Is it injective? Yes. Is it onto? No.

Let $f\colon \mathbb{Z} \to \mathbb{Z}$ be defined by $f(x) = x^2$. Is it injective? No. Is it onto? No.

Let $f\colon \mathbb{R}^+ \to \mathbb{R}^+$, where $\mathbb{R}^+$ denotes the positive real numbers, defined by $f(x) = x^2$. Is it injective? Yes. Is it onto? Yes.

Whether or not a function is injective has nothing to do with whether or not it is onto. All four possibilities can easily occur (of being one-to-one or not; of being onto or not).

Sometimes it can be tricky to figure out if a function is one-to-one or onto. Let $f(x) = 3x \bmod 10$ on $\mathbb{Z}_{10}$. Not one-to-one and not onto. But how about $g(x) = 3x \bmod 11$ on $\mathbb{Z}_{11}$. Now it is both one-to-one and onto.

A function that is both one-to-one and onto is called *bijective.* It is said to be a bijection. You can think of a bijection as a renaming of things. Every point in domain has an alternative name in the target. Every point in the target gets an alternative name in the domain.

A bijection $f\colon A \to A$ is called a *permutation.* It's again a renaming, but now you are renaming points within a set by points within that same set.

I sometimes like to think about the set of *all* functions from A to B. I denote this $\text{Func}(A, B)$. It comes up a lot in cryptography. Can we count $|\text{Func}(A, B)|$ when $A = B = \{0, 1\}^{128}$. Thinking about the truth table, it's $2^{128 \cdot 2^{128}} = 2^{2^{135}}$.

Similarly, let's write $\text{Perm}(A)$ for the set of all permutation on A. How big is $\text{Perm}(\{\mathtt{0}, \mathtt{1}\}^{128})$? Again thinking about the truth table, that will be $2^{128}!$.

Indexed families of permutations play a large role in cryptography. Perhaps we will have a chance to talk about it some, or explore it in a homework problem.

A wonderful thing about injective functions is that you can *invert* them on their range. That is, $f\colon A \to B$ with a range of C, then you can define a function $f'\colon C \to A$ by asserting that $f'(y)$ is the unique value x such that $f(x) = y$. You can't do this if f had a collision. If B *is* the range of f, that is, if f is onto and well as 1-to-1 (we called that a bijection), then $f^{-1}\colon B \to A$ is the *inverse* to f defined by $f(y) = x$ when x is the unique point in A such that $f(x) = y$.

Some example: Does the increment function on $\mathbb{Z}$ have an inverse. Yes, it's the decrement function. Does it have an inverse on $\mathbb{N}$? It does not, f was not onto, and there is nothing to map 0 to. Does the multiply by $c > 0$ function have an inverse on $\mathbb{R}$? It does, it's the multiply by $1/c$ function.

What about boolean negation? The function is its own inverse. And boolean conjunction? No, that's a map from $\mathbb{B}^2$ to $\mathbb{B}$, so the domain is bigger then the range and the function is not injective.

6 Computing Functions You Don't Know

Functions needn't have a simple domain or a simple description. Consider the function that takes as input the current configuration of a Go board and returns the move that the current version of MuZero (MZ) says it should move it. (MuZero is the successor of a program called AlphaGo that beat the reigning world champion, Lee Sedol, in Go.) It's a fairly complex function that spends many hours working before it is ready to process a single input. During those hours it creates a long string that will help it process subsequent board positions. So even if you understand quite well how it works abstractly, you certainly won't be able to make predications about input/output behavior of the algorithm after it has completed its self-learning phase. Not even if you are an expert Go player; the program will have surpassed you.

Or consider the Facebook News Feed function (FB), which decides what to send to your screen when you log on. You can think of it as mapping the ever-changing universe of data Facebook knows—the algorithm's *state*—along with whatever private information your computer sends to Facebook when you login, and returning what FB wants to print on your screen. It will also output an updated state for itself. And FB creates further changes to its state that are *not* user directed. The function is an ever evolving one with inputs from teams of people at the company, billions of users, and information mined from third parties.

While things like MZ (rather easily) and FB (less directly) can be regarded as functions, they are not functions that have definitions simpler than the code that computes them. These aren't exactly functions in the spirit of classical computer science, where we *knew* what function we wanted to compute and then tried to get a machine to compute it well. Instead, the computational artifact, the program itself, is the only real description of the function.

Let me broadly distinguish, then, among three sorts of functions I observe being computed these days:

1. In the *classical* computation of a function, you know what the function *is* that you are trying to compute. You can precisely describe it. The goal is find a good way to compute it—a nice *algorithm*. You will often need have to prove that your algorithm correct—prove that it computes the desired function. This is the sort of function and algorithm that an old-timer like me is happy with.

2. In the *playful* computation of a function, we create programs that compute *something*, but we don't know what exactly they compute. We can't characterize it in some crisp, alternative way. We explore the function created as a way to learn and have fun. I would put MZ squarely in this camp.

3. In the *power-seeking* computation of a function, everything is as immediately above—we can't cleanly characterize what we are computing—but the purpose is different. Now the purpose is to make somebody money or to tweak the world in somebody's interests. I would put FB squarely in this camp.

If the FB example of a "function" doesn't scare you, consider a DT function, for *dangerous-to-Trump*. It takes as input the string that represents everything known to the NSA. All emails people send, text messages, browsing queries, files stored on cloud servers, social media interactions. It outputs a *sentiment analysis* of each person in the world. Sentiments on whatever is of interest—here, say "feelings about Donald Trump". The sentiments might be understood as a lists of phrases, scored by strength. Perhaps the DT function also computes an influence score, measuring each individual's likelihood to influence others. A natural application of DT would be for someone powerful to disrupt the lives of anyone with very strong anti-Trump sentiments who is also to be highly influential. Disrupting change movements by targeting effectual individuals is a practice that the FBI has historically been engaged in.

Sounds like science fiction? I remember a faculty candidate whose research is pretty much what I've just described (but not specific to Trump, of course). Indeed sentiment-analysis is a major goal of NLP (natural language processing) and ML (machine learning).

Don't think functions that people want to compute are all like boolean AND or integer-addition-mod-2^{32}, say. Those are building blocks for more complex functions.

As I walked home yesterday I stopped to watch a black-and-white cat tentatively eying me. I stood still, hoping it would come over. When I finally approached, it backed away and continued its study. Thinking about today's lecture, I started wondering if the analysis the cat was making would be well understood as a function. Sensory input (encode them as strings, if you like) provide the cat with the raw data that it needs to interpret the world and make decisions like "slowly approach the old ape" or "back away from dangerous ape" or "lunge and try to make a meal of half-wit ape".

By the early 1940s this view of animals—including humans—as information-processing contraptions that compute complex functions from sensory input—this view had become a central part of the computer-science imagination. John van Neumann and Norbert Wiener were among the scientists strongly associated to it.

Cybernetics soon flourished in the Soviet Union, but it "crashed" in the U.S., becoming a term—even an aesthetic, and a literary and artistic genre—that few U.S. computer scientists wanted to be associated with.

Yet the viewpoint that humans and social systems are well understood and complex systems and functions subject to manipulation very much lives on. Facebook interventions *are* cybernetic explorations in information acquisition and social control. We've just stepped away from using the word.