Notes for ECE 369 - Discrete Mathematics for Computer Engineering

Ezekiel Ulrich

December 12, 2023

Contents

Course Introduction

This course introduces discrete mathematical structures and finite-state machines. Students will learn how to use logical and mathematical formalisms to formulate and solve problems in computer engineering. Topics include formal logic, proof techniques, recurrence relations, sets, combinatorics, relations, functions, algebraic structures, and finite-state machines. For more information, see the syllabus.

Equations

1. De Morgan's Theorem:

   $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$

   $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$

2. Modus ponens (mp)

   p

   $p \rightarrow q$

   $\therefore q$

3. Modus tonens (mt)

   $p \rightarrow q$

   $\neg q$

   $\therefore \neg p$

4. Predicate inference rules:

Name	Abrv.	Given	Can conclude
Existential generalization	eg	$P(a)$	$(\exists x)P(x)$
Existential instantiation	ei	$(\exists x)P(x)$	$P(a)$
Universal generalization	ug	$P(x)$	$(\forall x)P(x)$
Universal instantiation	ui	$(\forall x)P(x)$	$P(a)$

5. Propositional equivalence rules:

Expression	Equivalent to	Name - abbreviation
$p \vee q$ $p \wedge q$	$q \vee p$ $q \wedge p$	Commutative - comm
$(p \vee q) \vee r$ $(p \wedge q) \wedge r$	$p \vee (q \vee r)$ $p \wedge (q \wedge r)$	Associative - ass
$\neg(p \wedge q)$ $\neg(p \vee q)$	$\neg p \vee \neg q$ $\neg p \wedge \neg q$	De Morgan's Laws - De Morgan
$p \rightarrow q$	$\neg p \vee q$	Implication - imp
p	$\neg(\neg p)$	Double negation - dn
$p \leftrightarrow q$	$(p \rightarrow q) \wedge (q \rightarrow p)$	Def'n of equivalence - equ

6. Propositional inference rules:

From	Can derive	Name - abbreviation
$p, p \to q$	q	Modus ponens - mp
$p \to q, \neg q$	$\neg p$	Modus tollens - mt
p, q	$p \wedge q$	Conjunction - con
$p \vee q, \neg p$	q	Disjunction - dis
$p \wedge q$	p, q	Simplification - sim
p	$p \vee q$	Addition - add

Propositional Logic

We often wish that others would be more logical, tell the truth, or shower. While studying formal logic cannot help with the latter (in fact, studies have shown a negative correlation between hygiene and studying formal logic) it is a useful way to define what the first two mean. In a formal logic model, we have two constructs:

- **Statements/propostion**: A statement or a proposition is a sentence that is either true or false. Propositions are often represented with letters of the alphabet. For example: "q: the more time you spend coding, the less time you have to buy deodorant."
- **Logical connectives**: Used to connect statements. For example, "and" is a logical connective in English. It can be used to connect two statements, e.g. "the person next to me smells like dog *and* looks like a dog" to obtain a new statement with its own truth value.

  Here are common logical connectives in Boolean logic:

Logical Connective	**Symbol**
Negation (NOT)	$\neg$ or $'$
Conjunction (AND)	$\wedge$
Disjunction (OR)	$\vee$
Exclusive OR (XOR)	$\oplus$
Implication	$\to$
Biconditional	$\leftrightarrow$

Table 1: Connectives in Boolean Logic

Truth table: Defines how each of the connectives operate on truth values. Every connective has one. For example, consider $\wedge$ AND:

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Table 2: Truth table for $\wedge$

We see that p AND q is only true when p is true and q is true. Similarly, p or q is only true when p is true or q is true (or both). An important connective for discovering new truths is the implication $\rightarrow$, which basically says "if the first letter is true, then so is the second". Let p: "I live in Wiley" and q: "I have no AC". In English, the statement $p \rightarrow q$ would be stated as "If I live in Wiley, then I have no AC".

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Table 3: Truth table for $\rightarrow$

Table 3 shows the truth table for $\rightarrow$. It may not seem immediately clear why, for instance, if p and q are false, then $p \rightarrow q$ is true. If we consider what this means in English, then all we know is that I don't live in Wiley. Perhaps I live in Tarkington and still don't have AC, or perhaps I live in Honors and I do. In any case the first letter isn't true, so "if the first letter is true then so is the second" stands as true. If we have the statement $p \rightarrow q$, then we call q a *necessary condition* for p. Conversely, p is a *sufficient condition* for q.

Say we have a statement such as $A \vee B \rightarrow C$. This is ambiguous, since we can interpret it as either $(A \vee B) \rightarrow C$ or $A \vee (B \rightarrow C)$. The truth tables will differ in each case, so it becomes necessary to specify in what order we should apply logical connectives.

1. Parentheses "()"
2. Negation "$\neg$"
3. AND "$\wedge$"
4. OR "$\vee$"
5. Implication "$\rightarrow$"
6. Biconditional "$\leftrightarrow$"

Rules and proofs

With each additional variable in your truth table, the number of choices grows exponentially. Specifically, if you have n statement letters, you would have 2^n choices for your truth table.

Tautology: A formula that is true in every model. Example: I am president of the tautology club because I am president of the tautology club.

Contradiction: A formula that is false in every model. Examples: "it is raining and it is not raining", "I am sleeping and I am awake", "I am a good student and I attend IU".

Confusion often arises when negating a sentence such as "the book is thick and boring". An natural inclination is to negate it thus: "the book is not thick and not boring". However, consider the truth table for this: p: "the book is thick", q: "the book is boring". We can see the

p	q	$p \wedge q$	$\neg(p \wedge q)$	$\neg p \wedge \neg q$
T	T	T	F	F
T	F	F	T	F
F	T	F	T	F
F	F	F	T	T

last two rows are not identical, therefore the negation of "the book is thick and boring" is not "the book is not thick and not boring". For p to be false, either the book must not be thick *or* the book must not be boring. This is summarized by **De Morgan's Theorem**:

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

We now have a sufficient understanding of truth tables and logical connectives to come up with some useful rules. First of these are **Modus ponens (mp)**:

$$p$$
$$p \rightarrow q$$
$$\therefore q$$

and

Modus tollens (mt):

$$p \rightarrow q$$
$$\neg q$$
$$\therefore \neg p$$

Interestingly, it is possible to prove any statement in a system where a contradiction exists. This is known as the *principle of explosion*. To see how it works, consider the following example:

1. p: Donuts are good for you.
2. q: Unicorns exist.

I'll now assume the contradictory statement "donuts are good for you and donuts are not good for you".

$\neg p \wedge p$	(Given)
p	(1, simplification)
$p \vee q$	(2, addition)
$\neg p$	(1, simplification)
$\therefore q$	(3, disjunction)

Ergo, unicorns exist.

Below are two tables for commonly used rules.

Expression	Equivalent to	Name - abbreviation
$p \vee q$ $p \wedge q$	$q \vee p$ $q \wedge p$	Commutative - comm
$(p \vee q) \vee r$ $(p \wedge q) \wedge r$	$p \vee (q \vee r)$ $p \wedge (q \wedge r)$	Associative - ass
$\neg(p \wedge q)$ $\neg(p \vee q)$	$\neg p \vee \neg q$ $\neg p \wedge \neg q$	De Morgan's Laws - De Morgan
$p \to q$	$\neg p \vee q$	Implication - imp
p	$\neg(\neg p)$	Double negation - dn
$p \leftrightarrow q$	$(p \to q) \wedge (q \to p)$	Def'n of equivalence - equ

Table 4: Equivalence rules

From	Can derive	Name - abbreviation
$p, p \to q$	q	Modus ponens - mp
$p \to q, \neg q$	$\neg p$	Modus tollens - mt
p, q	$p \wedge q$	Conjunction - con
$p \vee q, \neg p$	q	Disjunction - dis
$p \wedge q$	p, q	Simplification - sim
p	$p \vee q$	Addition - add

Table 5: Inference rules

At this point, let us formally define an
Argument: An argument can be symbolized as

$$P_1 \vee P_2 \vee P_3 \vee ... \vee P_n \to Q$$

where P_i is called a hypothesis and Q is the conclusion. If this statement is a tautology, then the argument is *valid*. There are multiple ways we could prove a given argument is a tautology. For instance, we could create a truth table and brute force an answer. However, with even four hypotheses this process is tedious, and with each additional hypothesis it becomes exponentially harder. Therefore we instead often turn to the
Proof sequence: a sequence of well-formed formulas in which each formula is either a premise or the result of applying a derivation rule

to earlier well-formed formulas. In practice this looks like

P_1	(hypothesis)
P_2	(hypothesis)
P_3	(hypothesis)
	...
P_n	(hypothesis)
(formula) 1	(obtained from derivation rule)
(formula) 2	(obtained from derivation rule)
	...
(formula) n	(obtained from derivation rule)
$\therefore Q$	

Let's use all this new information in a simple proof.

A	(hypothesis)
$A \to B$	(hypothesis)
$B \to C$	(hypothesis)
B	(1, 2, mp)
C	(4, 3, mp)
$\therefore C$	

If we wish to apply our knowledge of logic to the real world, some practice in translating natural language to formal logic is necessary. Let's test it with this statement: "If chicken is on the menu, then don't order fish, but you should have either fish or salad. So if chicken is on the menu, have salad." Let C: "Chicken is on the menu", F: "You order fish", and S: "You have salad". We know that if chicken is on the menu you don't order fish, that you should have either fish or salad, and we'd like to show that if chicken is on the menu you should have salad.

C	(hypothesis)
$C \to \neg F$	(hypothesis)
$F \vee S$	(hypothesis)
$\neg F$	(1, 2, mp)
S	(3, 4, dis)
$\therefore S$	

Predicate logic

Predicate logic: Capable of making statements about entire groups instead of individual letters. In predicate logic, propositions are expressed in terms of predicates, variables and quantifiers, the latter of which propositional logic lacks.
Quantifer: How many objects have a certain property: "for every" or "for some".
Predicate: Property that a variable may have.
Domain of interpretation: Collection of objects from which the variable is taken.
Universal quantifer: "For all": $\forall$. States that a certain property holds for all objects in a domain.
Existential quantifer: "There exists": $\exists$. States that a certain property holds for at least one object in a domain.

As an example of a predicate well-formed formula: $(\forall x)[(\exists y)x > y]$. We would read this statment as "for all x there exists a y such that $x > y$." At first glance it may seem obvious that this statement is true, but consider the domain. What if the domain is all natural numbers? Then we could let $x = 1$ (or zero depending or your definition of natural numbers) and there would be no corresponding lesser y. We can see from this example that the truth value of a predicate logic formula depends on the domain as well as quantifiers and predicates.

Just as with propositional logic, we often need to translate English statements into predicate logic. Take the statement "every movie made by George Lucas is great". We can rephrase this as "for any movie, if the movie is made by George Lucas, it is great". We would write this formula as

$$(\forall x)(GL(x) \rightarrow Great(x))$$

(Author's note: no value judgement is associated with this English statement).

Let's examine some rules in predicate logic. First, negation:

$$\neg[\forall x A(x)] \leftrightarrow (\exists)x\neg A(x)$$

Some rules from propositional logic still apply in predicate logic. Take modus ponens as an example:

$$\begin{array}{rl} (\forall x)(\forall y)L(x,y) \rightarrow [(\exists x)H(x)] & \text{(hypothesis)} \\ \neg[(\exists x)H(x)] & \text{(hypothesis)} \\ \neg[(\forall x)(\forall y)L(x,y)] & \text{(1, 2, mt)} \\ (\exists x)(\exists y)\neg L(x,y) & \text{(3, DM)} \\ \hline \therefore (\exists x)(\exists y)\neg L(x,y) & \end{array}$$

Name	Abrv.	Given	Can conclude
Existential generalization	eg	$P(a)$	$(\exists x)P(x)$
Existential instantiation	ei	$(\exists x)P(x)$	$P(a)$
Universal generalization	ug	$P(x)$	$(\forall x)P(x)$
Universal instantiation	ui	$(\forall x)P(x)$	$P(a)$

Table 6: Predicate inference rules

Table 6 holds predicate inference rules. These rules hold given certain conditions. Namely:

(eg) x not in $P(a)$

(ei) Must the be first rule that introduces a

(ug) $P(x)$ not derived from a hypothesis with x as a free variable, and $P(x)$ is not derived by ei from wff with x as a free variable.

(ui) a is a constant.

A *free variable* is a variable not bound by a quantifer. For example, in the formula

$$(\forall x)(\forall y)P(x,y)$$

both x and y are bound by quantifers. Constrast this with the formula

$$(\exists x)(\forall y)q(x,y,z)$$

In this example, z is a free variable, since it is not associated with any quantifiers.

Let's see some of these rules in action by with a predicate logic proof. Say we have the statement "every ECE student works harder than somebody, and everyone who works harder than any other person gets less sleep than that person. Maria is an ECE student. Ergo, Maria gets less sleep than someone. Let $E(x)$: "x is an ECE student", $W(x,y)$: "x works harder than y", $S(x,y)$: "x gets less sleep than y", and m: Maria. We want to prove $\exists a(S(m,a))$.

$$\begin{aligned}
\forall x, E(x) \rightarrow (\exists y)(W(x,y)) &\quad \text{(hypothesis)}\\
\forall x, \forall y(W(x,y) \rightarrow S(x,y)) &\quad \text{(hypothesis)}\\
E(m) &\quad \text{(hypothesis)}\\
\exists y(E(m) \rightarrow S(m,y)) &\quad \text{(1, ui)}\\
E(m) \rightarrow W(m,a) &\quad \text{(4, ei)}\\
\forall y(W(m,y) \rightarrow S(m,y)) &\quad \text{(3, ui)}\\
W(m,a) \rightarrow S(m,a) &\quad \text{(6, ui)}\\
W(m,a) &\quad \text{(3,5 mp)}\\
S(m,a) &\quad \text{(7,8, mp)}\\
\exists a(S(m,a)) &\quad \text{(9, eg)}\\
\hline
\therefore \exists a(S(m,a))
\end{aligned}$$

Proofs

List of common proof techniques:

1. Exhaustive proof: In this kind of proof, the statement to be proved is split into a finite number of cases or sets of equivalent cases, and where each type of case is checked to see if the proposition in question holds
2. Refuting by counter-example: If we have a universal statement such as $\forall x(P(x) \rightarrow Q(x))$, we may show it to be false by finding a single a such that $\neg P(a)$.
3. Direct proof: Trying to prove $p \rightarrow q$, start by assuming p and then show q.
4. Proof by contraposition: The contrapositive of $p \rightarrow q$ is $\neg q \rightarrow \neg p$. A statement and its contrapositive are logically equivalent, so if proving $p \rightarrow q$ is too difficult we may try to prove $\neg q \rightarrow \neg p$ instead.
5. Proof by contradiction: Suppose I have to prove $p \rightarrow q$. I can begin by saying p is true and $\neg q$ is true. If by a series of steps I arrive a contradiction, then I may say p implies q.
6. Proof by induction: employs a neat trick which allows you to prove a statement about an arbitrary number n by first proving it is true when $n = 1$ (or some other base case), assuming it is true for $n = k$, and then showing it is true for $n = k + 1$. The steps to prove $\forall n P(n)$ are:
   (a) Prove $P(1)$ (this is your *base case*).
   (b) Assume for arbitrary $k \geq 1$, $P(k)$ (your *inductive hypothesis*).
   (c) Prove $P(k + 1)$.

Below are some example proofs using each of these techniques.

1. *Exhaustive proof (cases)*: say we wish to prove $|xy| = |x||y|$. Let's split this into four cases:
   (a) Case 1: x and y positive. Then the absolute values are equal to the original numbers, and we have

$$\begin{aligned} |x| &= x \\ |y| &= y \\ |xy| &= xy \\ |x||y| &= xy \\ \therefore |xy| &= |x||y| \end{aligned}$$

(b) Case 2: x and y negative. If x and y are both negative, then xy is positive. We thus have that

$$\begin{aligned} |x||y| &= xy \\ |xy| &= xy \\ \therefore |xy| &= |x||y| \end{aligned}$$

(c) Case 3: x negative and y positive. Now we have that xy is negative. Still, though, $|xy|$ will be positive (by def'n of $||$), and so will $|x|$ and $|y|$. So we again have that

$$|xy| = |x||y|$$

(d) Case 4: x positive and y negative. WLOG, case 3.

$$\therefore |xy| = |x||y| \quad \square$$

2. *Direct proof*: say we wish to prove the product of two even integers is even. We first need to translate this English sentence to a mathematical statement, which we can do in this case like so:

$$x = 2a, a \in \mathbb{Z}, y = 2b, b \in \mathbb{Z}, \rightarrow x \times y = 2c, c \in \mathbb{Z}$$

Our proof is below.

$$\begin{aligned} x &= 2a, a \in \mathbb{Z} \\ y &= 2b, b \in \mathbb{Z} \\ z &= x \times y \\ &= 2a \times 2b \\ &= 2(2ab) \\ &= 2c, c \in \mathbb{Z} \quad \square \end{aligned}$$

Since c is an integer, $2c$ is even and the proof is complete. Try proving the product of two odds is odd in a similar fashion.

3. *Proof by contradiction*: say we wish to prove $\sqrt{2}$ is irrational. In a theme that will become common as we see more proofs by contradiction, assume the opposite. That is, assume $\sqrt{2}$ is rational. By the definition of rational, we can then write

$$\sqrt{2} = \frac{a}{b}, a, b \in \mathbb{Z}$$

Where a and b share no common factors. We can then perform the following series of steps.

$$\begin{aligned} \sqrt{2} &= \frac{a}{b} \\ b\sqrt{2} &= a \\ 2b^2 &= a^2 \end{aligned}$$

This means that a^2 is even. It can be easily shown that if a^2 is even then a is even. That means that a^2 will actually be divisible by 4. We can rearrange to get

$$\begin{aligned} 2b^2 &= 4c, c \in \mathbb{Z} \\ b^2 &= 2c \end{aligned}$$

So b is likewise even. But if both a and b are even, then they share a common factor and our original supposition is false. Ergo $\sqrt{2}$ is irrational. $\square$.

4. *Poof by induction*: say we wish to show

$$\sum_{i=1}^{n} = \frac{n(n+1)}{2}$$

Begin with the base case $n = 1$.

$$\begin{aligned} \sum_{i=1}^{1} &= 1 \\ &= \frac{1(1+1)}{2} \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Since we have shown the base case to be true, we may now make our inductive hypothesis and assume that for arbitrary $k \geq 1$,

$$\sum_{i=1}^{k} = \frac{k(k+1)}{2}$$

Let us now show that the formula holds for $k+1$.

$$\begin{aligned} \sum_{i=1}^{k+1} &= \sum_{i=1}^{k} + (k+1) \\ &= \frac{k(k+1)}{2} + (k+1) \\ &= \frac{k(k+1) + 2(k+1)}{2} \\ &= \frac{k^2 + 3k + 2}{2} \\ &= \frac{(k+1)(k+2)}{2} \\ &= \frac{(k+1)((k+1)+1)}{2} \end{aligned}$$

And we are done $\square$.

The form of induction we have just seen is the *weak* form of induction. The *strong* form of induction is the following. To prove $\forall x, P(x)$,

The strong form is also known as the *second principle of mathematical induction*

we still prove the base case ($P(n)$). Now, however, we assume for arbitrary $k \geq 1$ that $P(r)$ is true for $1 \leq r \leq k$ and try to prove $P(k+1)$. Let's see this in action. Say we'd like to prove any postage greater than or equal to 8 cents can be created with a combination of 3 cent and 5 cent postage stamps. First, the base case. 8 can be created like so: $3+5=8$. Now let's assume for all $8 \leq r \leq k$, $P(r)$. Now the tricky part. To prove $P(k+1)$, notice that

$$\begin{aligned} P(k+1) &= k+1 \\ &= (k-2)+3 \\ &= (3a+5b)+3 \\ &= 3c+5b \quad \square \end{aligned}$$

We had to rewrite $k+1$ as $k-2+3$ so we could use our assumption that $P(k-2)$ is true. The astute among you will recognize that our proof is technically imcomplete. We have assumed $P(k-2)$, but what if we wish to prove $P(9)$? This is not included in our inductive step, since we only assume $P(r)$ for $8 \leq r \leq k$. We are in the domain of natural numbers (unless you have somehow managed to find a postage stamp with negative or fractional value), then there is no r for which this is true. Therefore we also need to prove $P(9)$ and $P(10)$, which is pretty simple.

We see that simply assuming $P(k)$ wouldn't be sufficient in this case, so there are proofs where we can use strong induction but not weak induction. Anything that can be proved with weak induction can be proved with strong induction, since in strong induction we assume $P(k)$ in addition to $P(r)$ for all r between 1 and k. Hence, "strong".

Let's see a more advanced example of induction. Say you have a set of n elements, and you want to create subsets of this set. You are interested in knowing how many subsets exist. After trying it out with $n = 1, 2, 3$ you suspect the number of subsets that can exist is 2^n, and you see that this problem is a good candidate for induction. Your base case is $P(1) = 2 = 2^1$, so that's out of the way. Now assume $P(k)$. That is, for any set with $k < n$ elements, the number of subsets is 2^k. You now need to show $P(k+1)$, which can be done by noticing that if we add an additional element to the set, then for every pre=existing group you can add the new element to get 2^k new groups, bringing your total number of subsets to $2^k + 2^k = 2 \times 2^k = 2^{k+1}$, and we are done.

We've seen some examples of when induction is used well, but doing everything perfectly is tiring. Let's show something false: any group of horses are all the same color. Obviously the base case is true: a group of one horse is always the color of that horse. Assume $P(k)$, that any group of k horses is monochromatic. Now we need to prove it for $k+1$. First, exclude one horse and look only at the other k horses; all these are the same color, since k horses always are the same color. Likewise, exclude some other horse (not identical to the one first removed) and look only at the other k horses. By the same reasoning, these, too, must also be of the same color. Therefore, the

first horse that was excluded is of the same color as the non-excluded horses, who in turn are of the same color as the other excluded horse. Hence, the first horse excluded, the non-excluded horses, and the last horse excluded are all of the same color, and we are done. What's the issue here? The issue is that in order to select two different horses, we need at least two different horses. However, our base case was with one horse. To use this logic we would need to show $P(2)$, for which our original argument is obviously not true.

Proofs of correctness

Say you are part of a team developing software for the NSA or NASA. A common requirement for the deliverable is to prove that it meets certain properties. You may be asked to show that your code will eventually return an answer (i.e. the algorithm terminates) or that, *if* an answer is returned, it will be correct. Correct in this sense means your program meets whatever specifications have been laid out.

Broadly speaking, there are two properties that a program must satisfy. The first is **safety**: the program will not violate any invariant that you write (think `assert` for C programs. If you always need to return a positive integer, for instance, and you can show that it always does, then you have satisfied safety). The second is **liveness**: that your program is going to terminate. This can be done by showing there are no infinite loops, for example. You may also specify the amount of time in which your program is guarenteed to terminate. In the field, the longest possible time your program will take is known as Worst Case Execution Time (WCET).

In software development, as you are likely aware, your code is often checked using tests. This is one method of gaining evidence that the kernel of your program is correct. We may also prove that the code is correct, which is often more laborious. Unless you're developing software for safety-critical areas, you'll probably use testing more than formal proof. If you needed to unequivocally show that your program will do what it should under all circumstances of interest, you'll have to use formal logic techniques.

Note that "testing can prove the presence of errors but never their absence". We cannot show through testing that a program will behave exactly as it should in all cases, unless the number of possible inputs is so small that it is possible to enumerate through them. Even though testing cannot prove the correctness of a program, it can still reveal issues and build confidence that the code is correct.

To understand how we can apply logical rules to programming, imagine your algorithm as a function P that takes in some input values X and produces some output values Y.

$$Y = P(X)$$

The predicate $Q(X)$ describes conditions that the input values will satisfy (e.g. $X > 0$, $X \in \mathbb{R}$, X is a string under 127 characters). The predicate $R(X, Y)$ describes conditions that the output must satisfy for a given input (e.g. $Y = \sqrt{X}$, Y gives the prime factorization of X, Y capitalizes X). We say a program is correct if the implication

$$\forall X[Q(X) \rightarrow R(X, P(X))]$$

is valid. That is, whenever $Q(X)$ is true of the inputs, $R(X, Y)$ should be true of the outputs. The notation changes when dealing with program correctness: here's the more common way to write the above implication:

$$\{Q\}P\{R\}$$

The terminology here is that Q is a *precondition* for program P, and R is the *postcondition*.

Precondition: a condition that is true before the execution of a program.

Postcondition: a condition that is true after the execution of a program.

A program is broken down into individual statements s_i, with predicates sandwiching them. Here is the general form of a predicate-statement sandwich:

$$\begin{aligned} &\{Q\} \\ &s_0 \\ &\{R_1\} \\ &s_1 \\ &\{R_2\} \\ &\dots \\ &s_{n-1} \\ &\{R\} \end{aligned}$$

$Q, R_1, R_2, R_n = R$ are assertations. Your program P is provably correct if each of the following implications holds:

$$\begin{aligned} \{Q\}s_0\{R_1\} \\ \{R_1\}s_1\{R_2\} \\ \dots \\ \{R_{n-1}\}s_{n-1}\{R\} \end{aligned}$$

So to prove correctness for P, "all" you need to do is produce this sequence of valid implications.

With the idea of correctness hopefully clear, let me introduce you to *assignment statements*. An assignment statement is something with an equals, like $x = a$. Assignment statements often come with a postcondition. It will be your job to find the precondition that makes it true. For example, say you have the assignment statement $y = x + 1$ and the postcondition $y = 10$. What is the precondition that makes this true? What is the Hoare triple?

The appropriate rule of inference for assignment statements is the **assignment rule**: states that $\{R_i\}s_i\{R_{i+1}\}$ is valid provided s_i is an assignment statement ($x = a$) and R_i is R_{i+1} with a substituted everywhere for x.

Let's put all of this into practice with an example. Say we wish to prove the following computes $x(x-1)$ correctly.

Listing 1: Assignment rule example

$\{Q\}P\{R\}$ is called a *Hoare triple* and gives before and after conditions for a program fragment. For instance, precondition: $Q(x)$, program: $Y = P(x)$, postcondition: $R(x, y)$ is a Hoare triple that means

$$(\forall X)Q(X) \to R(X, Y)$$
$$(\forall X)Q(x) \to R(X, P(X))$$

These sandwiching predicates are also called *assertations*, because having multiple names for the same concept makes it much more fun to learn.

In proving code correctness, we often put arithmetically equivalent assertations in sequences with no lines of code in between, like

$$\begin{aligned} \{y = 4\} \\ \{y + 10 = 14\} \\ x = y + 10 \\ \{x = 14\} \end{aligned}$$

```
y = x - 1
y = x * y
```

To show this, we can use the assignment rule with `y = x - 1` and plug it into `y = x * y` to get `y = x * (x - 1)`. The key principle here is that you can use proof rules to show that a postcondition holds for a given set of preconditions. As long as the given set of preconditions is a subset of the derived preconditions, you're good and the postconditions will be met.

Let's examine conditional statements now. Proving a conditional statement "$\{Q\}$ if B then P_1 else P_2 $\{R\}$" boils down to showing two things:

1. $\{Q \text{ and } B\}\ P_1\ \{R\}$
2. $\{Q \text{ and } \neg B\}\ P_2\ \{R\}$

Say we have something like the following:

Listing 2: Example

```
{x = 7}
    if x <= 0
        y = x
    else
        y = 2*x
{y = 14}
```

We must show each of the two cases, that

1. $\{x = 7 \wedge x \leq 0\}\ y = x$
2. $\{x = 7 \wedge x > 0\}\ y = 2x$

We can use the assignment rule with `x = 7` to show this is true. In case 2, we have $y = 2 \times 7 = 14$, so our postcondition is true. In case 1, x is not less than or equal to zero, so we have that y is true by the defn of implication. We have then shown that both are cases are true and we are done.

Let's see another example of a conditional proof. We want to verify the correctness of this code block:

Listing 3: Example

```
{x = 11}
    y = x - 1
{y = 10}
    if x <= 0
        z = y - 1
    else
```

```
            z = y + 3
{z = 13}
```

We must show each of the two cases again.

1. $\{y = 10 \wedge y \leq 0\}\ z = y - 1\ \{z = 13\}$
2. $\{y = 10 \wedge y > 0\}\ z = y + 3\ \{z = 13\}$

Again, since $10 \not\leq 0$, the first case is true. For the second, use the assignment rule.

$$
\begin{aligned}
z = 13 z & \qquad = y + 3 \\
13 = y + 3 & \text{ by assignment rule} \\
y = 10 & \\
y = x - 1 & \\
10 = x - 1 & \text{ by assignment rule} \\
x = 11 &
\end{aligned}
$$

So in the second case, our derived precondition `x = 11`, is a subset of our given precondition, so it works here as well.

Let's look at something a lot more interesting: loop statements.

Listing 4: Loop

```
while B
    S
```

So while B is true, the program will do S. If B ever becomes false then the program stops. We could repeatedly perform statement S until B is false,

$$
\begin{aligned}
\{Q\}\{R\} \\
\{Q\}S\{R\} \\
\{Q\}S;S\{R\} \\
\ldots
\end{aligned}
$$

covering every case: that the loop executes no times, once, etc, but this would be exhausting, so let's use another method instead. We must find a *loop invariant*, a statement that is true no matter how many times the loop executes. Then we must show that the loop invariant and not B implies the conclusion we want to verify. Specifically, since we want to prove the implication

$$\{Q\}s_i\{R\}$$

we should find a loop invariant Q such that

$$\{Q\}s_i\{Q \wedge \neg B\}$$

Listing 5: Loop invariant

```
Sum(n)
i = 1;
j = 0;
while i != n
    j = j + i
    i = i + 1
```

What's something that's true here no matter the number of times the loop executes? How about j = 0 + 1 + ... + (i - 1)? Since this program is supposed to calculate the sum from 0 to n - 1, this would be a useful thing to prove. To show this is true, we can use induction. The base case is when the loop executes not at all, so j = 0 and i = 1. It is definitely true that 0 = 1 - 1, so the base case works. Now we assume jk = 0 + 1 + ... (ik - 1), and try to show that j(k+1) = 0 + 1 + ... (i(k+1) - 1). Try this yourself: it is a good exercise in induction. Once we have proved this, then we have to show that the loop invariant and not B (i = n) gives us the desired result, which in this case is the sum from 0 to n - 1.

Let's try another example.

Listing 6: Loop invariant example

```
{a >= b, not both zero}
GCD(a, b)
i = a;
j = b;
while j != 0
    r = i mod j
    i = j
    j = r
{i = gcd(a,b)}
```

Here's the loop invariant: gcd(i,j) = gcd(a,b). We know $\neg B$ is j = 0. Therefore, gcd(i,0) = gcd(a,b) = i. Now we just need to prove that the loop invariant is true, again usign induction. The hardest part of these problems is coming up with a loop invariant, after that step we just apply induction. If you can find a loop invariant that's useful to you and is actually true after an abritrary number of executions, you are basically done.

Here's another example that computes the nth power of 2.

Listing 7: 2**n

```
i = 1
j = 2
```

```
while i != n
    j = j * 2
    i = i + 1
return j //the power of 2
```

Here's the loop invariant: $j = 2^i$. When this loop executes 0 times, we have that $i = n = 1 \wedge j = 2 = 2^1$, so the base case is valid. Now, we need to assume $P(k)$: $j_k = 2^{i_k}$ and show $P(k+1)$: $j_{k+1} = 2^{i_{k+1}}$. From the program, we know that

$$\begin{aligned} j_{k+1} &= 2 * j_k \\ i_{k+1} &= i_k + 1 \end{aligned}$$

Ergo,

$$\begin{aligned} j_{k+1} &= 2 * 2^{i_k} \\ &= 2^{i_k+1} \end{aligned}$$

And so it is proven.

Recurrences

As you have likely seen, recursions consist of two parts.

1. Basis step: define something simple not in terms of itself.
2. Recursive step: define new cases in terms of previous cases

What is the basis step for the Fibonacci sequence? What is the recursive step?

We are not limited to sequences for recursion. We may also have sets, or other constructs. Sometimes, it is possible to find an explicit formula for the nth element of a recursively defined sequence: this is called the *closed form*.

In computer programming, we are especially interested in recursive operations. For example, say we have the function `exp(x,n)`, $n > 0$. We can define this recursively. Our basis step will be $n = 1$, where $x^n = x$. Now we need the inductive step, which will be `exp(x,n) = x * exp(x, n-1)`. Recursion and closed form is constrasted with iteration (loops), and often multiple of the three methods are possible to solve a problem. To find the closed form, our best approach is often to simply look at the first few terms of the sequence, try to spot a pattern, and then prove that pattern holds for any term. In class, this is called the *expand-guess-verify* approach.

Just because any method could be used doesn't mean they are equivalent. Recursive solutions are often more succinct, but use more memory because each time the function is called a new frame is pushed on the stack. Iterative solutions are usually longer to define, but use less memory. Closed forms are the best, if you can find one.

We don't always need to use this approach, sometimes the solution for a certain form of recurrence is known. A relation of the form

$$S(n) = cS(n-1) + g(n)$$

is known as a *linear first order* relation. It can be shown that the

If $g(n) = 0$, then the relation is also *homogeneous*.

solution to such a relation is given by

$$S(n) = c^{n-1}S(1) + \sum_{i=2}^{n} c^{n-i}g(i)$$

For example, say we have the recurrence

- $T(1) = 1$
- $T(n) = T(n-1) + 3, n \geq 2$

By the above formula, we have that the closed form is given by

$$\begin{aligned} T(n) &= c^{n-1}T(1) + \sum_{i=2}^{n} c^{n-i}g(i) \\ &= 1^{n-1}T(1) + \sum_{i=2}^{n} 1^{n-i}3 \\ &= T(1) + \sum_{i=2}^{n} 3 \\ &= 1 + \sum_{i=2}^{n} 3 \\ &= 1 + 3(n-1) \\ &= 3n - 2 \end{aligned}$$

As another example, say we have

- $A(1) = 1$
- $A(n) = A(n-1) + n^2, n \geq 2$

Again using the formula, we'll have

$$\begin{aligned} A(n) &= c^{n-1}A(1) + \sum_{i=2}^{n} c^{n-i}g(i) \\ &= 1^{n-1}A(1) + \sum_{i=2}^{n} 1^{n-i}i^2 \\ &= A(1) + \sum_{i=2}^{n} i^2 \\ &= 1 + \left(\frac{n(n+1)(2n+1)}{6} - 1\right) \\ &= \frac{n(n+1)(2n+1)}{6} \end{aligned}$$

If you have a *second order homogeneous recurrence relation*, which is of the form

$$S(n) = c_1S(n-1) + c_2S(n-2),$$

then the closed form is different. Say the characteristic equation

$$t^2 - c_1t - c_2 = 0$$

has two roots r_1 and r_2. Then

$$S(n) = pr_1^{n-1} + qr_2^{n-1}.$$

If the two roots are not distinct, then

$$S(n) = pr^{n-1} + q(n-1)r^{n-1}.$$

In either case p and q may be found by substiting values of n into the formula and solving the resultant system of equations. Say we have the Fibonacci sequence, defined by

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 2 \\ F(n) &= F(n-1) + F(n-2). \end{aligned}$$

Then the polynomial we need to solve is

$$t^2 - t - 1 = 0.$$

The solutions are given by the quadratic formula,

$$\begin{aligned} t &= \frac{1 \pm \sqrt{1+4}}{2} \\ &= \frac{1 \pm \sqrt{5}}{2}. \end{aligned}$$

Call the two roots ϕ and ψ. Then the solution is

$$\begin{aligned} F(n) &= p\phi^n + q\phi^n \\ F(0) &= p + q \\ &= 1 \\ F(1) &= p\phi + q\psi \\ &= 1 \end{aligned}$$

Solving the system of equations yields

$$\begin{aligned} p &= \frac{1}{\sqrt{5}} \\ q &= \frac{1}{\sqrt{5}}. \end{aligned}$$

So our final formula is

$$F(n) = \frac{\phi^n - \psi^n}{\sqrt{5}}.$$

It is natural to wonder what recurrence relations have an equivalent closed form, and how to find that closed form. Let's start with a simple example. Say we have a *first-order homogeneous linear* recurrence

relation of the form

$$S(1) = S_0$$
$$S(n) = cS(n-1)$$

It is easy to see that the solution is

$$S(n) = S_0 c^{n-1}.$$

Take instead a quadratic homogeneous relation of the form

$$S(1) = S_0$$
$$S(n) = cS^2(n-1).$$

Hint: try calculating $S(n)$ by plugging in $S(n-2), S(n-3), \ldots$

Then what will the closed form look like? Say you have a kth order linear homogeneous relation of the form

$$S(n) = c_1 S(n-1) + c_2 S(n-2) + \cdots + c_k S(n-k).$$

The characteristic equation will be

$$t^k - c_1 t^{k-1} - \ldots c_{k-1} t - c_k.$$

If the ith root is r_i, then

$$S(n) = \alpha_1 r_1^{n-1} + \alpha_2 r_2^{n-1} + \cdots + \alpha_k r_k^{n-1}.$$

This formula implies that for second, third, and fourth order recurrence relations it is always possible to find a closed form, since one can always find the roots for second, third, and fourth degree polynomials. However, unlike quadratics, cubics, and quartics, the general fifth-or-above order polynomial cannot be solved as proved by the Abel-Ruffini theorem.

Combinatorics

A set is a collection of distinct objects. There can be any number of items, be it a collection of whole numbers, months of a year, types of birds, and so on. Each item in the set is known as an element of the set. Some examples of sets:

$$A = \{1, 2, 3, 4, \ldots\}$$
$$S = \{"blue", "red", "green", 10, 10, 20\}$$
$$Z = \{\frac{\pi}{2} n\}, n \in \mathbb{Z}^+$$

Table 7 holds common set operations and notation.

Sets are often associated with combinatorics. The foundational idea of combinatorics is known as the *fundemental princple of counting*, and says if an event can occur in m different ways, and another event can occur in n different ways, then the total number of occurrences of the events is mn.

Name	Description	Symbol
Union	All elements in either A or B	$A \cup B$
Intersection	Only shared elements in A and B	$A \cap B$
Subset	A is composed only of elements in B	$A \subseteq B$
Proper subset	A is a subset of B and $A \neq B$	$A \subset B$
In	x is a member of A	$x \in A$
Cartesian product	$\{(a,b)\|x \in A \wedge y \in B\}$	$A \times B$
Squaring	$A \times A$	A^2
Empty set	The set with no elements	$\varnothing$

Table 7: Set notation

We also have the principle of inclusion and exclusion, which states that the number of elements in $A_1 \cup A_2 \cup \cdots \cup A_n$ is

$$\begin{aligned} N &= |A_1 \cup A_2 \cup \cdots \cup A_n| \\ &= \sum_{1 \leq j \leq n} |A_i| - \sum_{1 \leq i < \leq n} |A_i \cap A_j| \\ &+ \sum_{1 \leq i < j < k \leq n} |A_i \cap A_j \cap A_k| - \ldots \\ &+ (-1)^{n+1} |A_i \cap A_j \cap \cdots \cap A_n|. \end{aligned}$$

That's a nasty formula. What it's saying is when you have the union of sets, you can't just add the number of elements in each together because they might share elements, and sets don't allow duplicates. So you have to subtract their intersection to avoid overcounting. If you have just two sets, the formula becomes

$$N = |A| + |B| - |A \cap B|,$$

which hopefully is slightly clearer.

With all this talk of combinatorics, I would be remiss to not define a permutation. A permutation is an ordered arrangement of objects. Contrast this with a combination, where order does not matter. We denote a permutation of k distinct objects from n total objects as $P(n,k) = \frac{n!}{(n-r)!}$. The formula for a combination (where order doesn't matter) is $C(n,k) = \frac{n!}{(n-k)!k!}$ Note that we are assuming each object is distinct. If the first object is repeated r_1 times, the second r_2 times, and so on, out of a total of n objects, then the number of possible permutations is $\frac{n!}{r_1!r_2!\ldots r_n!}$.

Now, let's turn to a favorite of mathematical problem solvers, the *pigeon hole principle*. The principle states that if more than k pigeons fly into k pigeonholes, then at least one pigeonhole will have more than one pigeon. More generally, If more than k items are placed into k bins then at least one bin will contain more than one item. This seemingly trivial principle has surprisingly deep ramifications for

See also the Chicken McNugget theorem, which states that states that for any two relatively prime positive integers m, n, the greatest integer that cannot be written in the form $am + bn$ for nonnegative integers a, b is $mn - m - n$.

problem solving. The pigeon hole principle can answer questions such as "how many times must a die be rolled to guarentee that the same number is obtained twice?".

Combinatorics are used in the binomial theorem, which states that

$$(a+b)^n = C(n,0)a^n + C(n,1)a^{n-1}b + C(n,2)a^{n-2}b^2 + C(n,j)a^{n-j}b^j + C(n,n)b^n$$
$$= \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

I am fond of using Pascal's triangle when computing coefficients by hand instead. Not only is it faster and easier, it also has interesting properties by itself.

k							
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1
	0	1	2	3	4	5	6
				i			

Relations

The simplest form of mathematical relationship is the binary relation. Recall the Cartesian product of a set with itself is $S \times S = \{(x,x)|x \in S\}$. A binary relation is just a subset of $S \times S$. For example, say we have the set $S = \{1,2,3\}$ and we are interested in the equality relation. It's common to give the relation as a predicate instead of enumerating all the elements in the set. As an example, let the relation ρ be that $x\rho y \leftrightarrow x = y$. Then $\rho = \{(1,1),(2,2),(3,3)\}$. Notice that $\rho \subset S \times S$. The size of $S \times S$ will be n^2, since you have n choices for the first element and n for the second. The number of possible binary relations is the number of subsets of $S \times S$, so the number of possible binary relations is 2^{n^2}.

Sorry, poly folks.

Because mathematicians love looking impressive, Greek letters are often used for relations. ρ is common.

If you have a binary relation ρ that consists of a set of ordered pairs (s_1, s_2), then there are four possibilities.

1. One-to-one relation: Each s_1 is paired with more than one s_2.
2. One-to-many relation: At least one s_1 is paired with more than one s_2.
3. Many-to-one relation: At least one s_2 is paired with more than one s_1.

4. Many-to-many relation: At least one s_1 is paired with more than one s_2 and at least one s_2 is paired with more than one s_1.

Note that not all elements in the set need to appear in ρ. This sets relations apart from functions.

In some situations, we may with to perform operations on relations. Let ther be two binary relations ρ and σ on a set S. Each is a subset of $S \times S$ and we can perform set operations on them to create new subsets, which represent new binary relations. For example, say we have that

$$x\rho y \leftrightarrow x = y$$
$$x\sigma y \leftrightarrow x < y$$

$\rho \cup \sigma$ is simply given by

$$x(\rho \cup \sigma)y \leftrightarrow x \leq y.$$

Relations may have properties. For example, $x\rho y \leftrightarrow x = y$ has these properties:

1. $\forall x \in S, (x, x) \in \rho$.
2. $\forall x, y \in S, (x, y) \in \rho \rightarrow (y, x) \in \rho$.
3. $\forall x, y, z \in S, (x, y) \in \rho \wedge (y, z) \in \rho \rightarrow (x, z) \in \rho$.

Each of these properties has a name, and you are likely familar with at least some of them.

1. Reflexive: $(\forall x)(x \in S \rightarrow (x, x) \in \rho)$.
2. Symmetric: $(\forall x)(\forall y)(x \in S \wedge y \in S \wedge (x, y) \in \rho \rightarrow (y, x) \in \rho)$.
3. Transitive: $((\forall x)(\forall y)(\forall z)(x \in S) \wedge (y \in S) \wedge (z \in S) \wedge (x, y) \in \rho \wedge (y, z) \in \rho \rightarrow (x, z) \in \rho)$.

There is an opposite of the symmetric property, called anti-symmetric. Anti-symmetry means that

$$(\forall x)(\forall y)(x \in S \wedge y \in S \wedge (x, y) \in \rho \wedge (y, x) \in \rho \rightarrow x = y),$$

or basically that for any (x, y) where x and y are distinct, ρ does not contain (y, x).

Consider the relation $\leq$. What properties does this relation have? Let's go through the properties.

1. Reflexive: yes, since $x \leq x$.
2. Symmetric: no, since it is not true that if $y \leq x$ then $x \leq y$.

3. Transitive: yes, since if $x \leq y$ and $y \leq z$ then $x \leq z$.

4. Anti-symmetric: yes, since if $(y, x) \in \rho$ and $(x, y) \in \rho$ then $x = y$.

If a relation fails to have a property, for example reflexivity, then finding the closure means finding the smallest number of elements to add to the relationship set such that it holds the desired property. For instance, say you have $\rho = \{(1, 2)\}$ and want to find the symmetric closure. The smallest number of elements we can add to this to close it is one, to get $\rho^* = \{(1, 2), (2, 1)\}$.

There is an algorithmic process for finding the transitive closure, but such things can be done by intuition in most cases. For instance, say we wish to find the closure of $\rho = \{(9, 3), (5, 11), (5, 12), (5, 3), (3, 4)\}$ under transitivity. We can see that anytime we have an element (a, b), then we need to look and see if there is any (b, c), and if there is, ensure that (a, c) is also in ρ. Doing it for the ρ in this case, we have that

$$\rho^* = \{(9, 3), (5, 11), (5, 12), (5, 3), (3, 4), (9, 4), (5, 4)\}.$$

Simple enough.

If you really wish to find it via algorithm, the usual method is to write out a matrix A where $(i, j) \in \rho \to A[i, j] = 1$. Then write out $A^{(2)}$, which has a 1 if j can be reached in two steps from i. Now $A^{(3)}$, $A^{(4)}$, and so on, then $\wedge$ all the matrices together. This will give us a matrix representing what elements can be reached from other elements in any number of steps. The final thing to do is transform the matrix back into a relation.

Let's now look at *partial* and *total orderings*.

Partial ordering: a binary relation on S which is reflexive, anti-symmetric, and transitive. Some examples of partial orderings on $\mathbb{N}$ are $x \leq y$ and $x|y$.

Predecessor of y: if $x \rho y \wedge x \neq y$, x is the predecessor of y (denoted $x < y$).

Immediate predecessor of y: if $x \rho y \wedge x \neq y \wedge \neg(\exists z, x < z < y)$, then x is the immediate predecessor of y (denoted $x < y$).

(S, ρ) is called the partially ordered set, or *poset*.

Total ordering: A partial ordering in which every element is related to every other element. For example, the relation $\leq$ on the set of reals.

There are some special elements in the poset. There is the greatest element, the least element, the maximal element, and the minimal element.

Greatest element: if $y \in S \wedge (\forall x \in S, x < y)$ then y is the greatest element. Basically, y must be greater than all other elements.

Maximal element: $y \in S$ is maximal if there does not exist any element $x \in S$ such that $y < x$.

Least element: if $y \in S \wedge (\forall x \in S, x > y)$ then y is the least element. That is, y must be lesser than all other elements.

Minimal element: $y \in S$ is minimal if there does not exist any element $x \in S$ such that $y > x$.

Here is a joke: There are 10 kinds of people in the world: those who understand binary, those who don't, and those who understand ternary.

There's another special kind of relation to add to partial and total orderings, the *equivalence relation*.

Equivalence relation: a binary relation on S which is reflexive, symmetric, and transitive. For example, $\mathbf{N}, \rho = x + y = 2n, n \in \mathbf{Z}$ (all

natural numbers x and y such that $x + y$ is even.) We can make a subset of S where all the members are equivalent to one another, called an equivalence class. In the previous example, the two equivalence classes would be all evens or all odds, since any two numbers of the same parity summed will be even.

A special equivalence relation is congruence modulo n ($x \mod n$). Let's first show why this is an equivalence relation. We require it to be reflexive, symmetric, and transitive. It is certainly reflexive: $x \mod n \cong x \mod n$. For symmetry, if $x \mod n \cong y \mod n$ then $y \mod n \cong x$, so it is symmetric. The transitive case is left as an exercise to the reader. The equivalence classes would be $a \in \mathbf{Z}, x|x = na, x|x = na + 1, \ldots, x|x = na + (n - 1)$, since each of these is equivalent under modulo congruence.

Functions

We can also define a *function* between two sets S and T, where every element S maps to one and only one element in T. We call S the domain, and T the co-domain. If (s, t) belongs to the function, t is the image of s, and s is the pre-image of t. The *range* of f is the set R of images of all members of S. If the range of f is identical to the co-domain, then it is an *onto* function. If a function is both one-to-one and onto, then it is *bijective*.

Bijective functions have inverses. In fact, if the inverse of f exists, then f is a bijection.

Functions can be composed. If you have a function f from S to T and another function g from T to U, then for any $s \in S$, $f(s) \in T$. Threfore, $g(f(s)) \in U$. In other words, $g(f(s))$ is a new function from S to U. $g(f(x))$ is called the composition function. For composing, the domains and ranges have to be compatible. That means that you can't commute f and g willy-nilly; $f(g(s)) \neq g(f(s))$. s might not even be in the domain of g. Order matters! Actually, the only time when $f(g(x)) = g(f(x))$ is when f and g are inverses. Then, $f(g(x)) = x$ and $f(g(x))$ is the *identity function*.

Say we have f such that f maps from S to T. If $|S| = m$ and $|T| = n$, then there are n^m possible functions (not necessarily one-to-one). If the function must be one-to-one, then $m \leq n$ and the number of functions is $\frac{n!}{(n-m)!}$. The number of onto functions is the total number of functions minus the number of non-onto functions, or

$$n^m - [\binom{n}{1}(n-1)^m - \binom{n}{2}(n-2)^m - \cdots - \binom{n}{n-1}(1)^m].$$

For a given set A, consider bijective functions from A to A. These functions are called permutations of A. For example, if $A = \{1, 2, 3, 4\}$, then a function might map 1 to 2, or to 3, or to 4. Then 2 could map to any of the three remaining, and so on for a total of $n!$ possible permutations.

Less commonly seen are *derangements*. Derangements are permutations such that no element is mapped onto itself.

A useful thing to know is the *order* Θ of a function, which can tell you roughly how fast it executes. We say functions f and g are of the same order if

$$c_1 g(x) \leq f(x) \leq c_2 g(x), \forall x \geq n_0.$$

If that is the case, we would say $f = \Theta(g)$. We say that f is *big O* of another function g if

$$f(x) \leq cg(x), \forall x \geq n_0.$$

If that is the case, we would say $f = O(g)$.

To prove either O or Θ, we simply need to choose $c_{(i)}$ to satisfy the requirements.

If $f = \Theta(g)$ then they are in the same equivalence class with respect to runtime.

Finite state machines

A *finite state machine* (FSM) is a model of a computer. It has the following characteristics:

- Has a discrete clock
- Deterministic
- Takes inputs from a given alphabet
- Generates outputs from given alphabet
- In one of a finite number of states
- Next state is determined by current state and input

Say we have $M = [S, I, O, f_s, f_o]$ as the FSM. S is a finite set of states, with a given start state s_0. I and O are input and output alphabets respectively. $f_s : S \times I \to O$, and $f_o : S \to O$.

Finite state machines are commonly represented using state graphs, such as fig. 1. We can represent this using a state table.

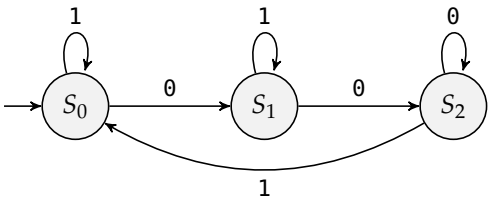

Figure 1: State graph

Output state	Input = 0	Input = 1	Output
S_0	S_1	S_0	0
S_1	S_2	S_1	1
S_2	S_2	S_0	0

FSMs can be used as a *recognizer*, a machine that outputs a 1 when the input string matches a certain description. Formally, there is one starting state. You feed it an input σ which is a subset of all the possible strings you can create from the input alphabet. Then, the FSM outputs a 1 and we say it accepts the string.