Algorithm redundancy

Real-life representations are often prone to corruption. Biological codes, like RNA, may mutate naturally[1] and during measurement; cosmic radiation and other ambient noise can flip bits in computer storage[2]. One way to recover from corrupted data is to introduce or exploit redundancy.

Consider the following algorithm to introduce redundancy in a string of 0s and 1s.

Create redundancy by repeating each bit three times

```
procedure redun3(a_{k-1} ··· a_0: a nonempty bitstring)
for i := 0 to k-1
  c_{3i} := a_i
  c_{3i+1} := a_i
  c_{3i+2} := a_i
return c_{3k-1} ··· c_0
```

Decode sequence of bits using majority rule on consecutive three bit sequences

```
1 procedure decode3(c_{3k-1} ··· c_0: a nonempty bitstring whose length is an integer multiple of 3)
2 for i := 0 to k-1
3   if exactly two or three of c_{3i}, c_{3i+1}, c_{3i+2} are set to 1
4     a_i := 1
5   else
6     a_i := 0
7 return a_{k-1} ··· a_0
```

Give a recursive definition of the set of outputs of the *redun*3 procedure, *Out*,

Consider the message $m = 0001$ so that the sender calculates $redun3(m) = redun3(0001) = 000000000111$.

Introduce ____ errors into the message so that the signal received by the receiver is ____________ but the receiver is still able to decode the original message.

Challenge: what is the biggest number of errors you can introduce?

Building a circuit for lines 3-6 in *decode* procedure: given three input bits, we need to determine whether the majority is a 0 or a 1.

c_{3i}	c_{3i+1}	c_{3i+2}	a_i
1	1	1	
1	1	0	
1	0	1	
1	0	0	
0	1	1	
0	1	0	
0	0	1	
0	0	0	

Circuit

[1] Mutations of specific RNA codons have been linked to many disorders and cancers.

[2] This RadioLab podcast episode goes into more detail on bit flips: `https://www.wnycstudios.org/story/bit-flip`

Cartesian product definition

Definition: The **Cartesian product** of the sets A and B, $A \times B$, is the set of all ordered pairs (a, b), where $a \in A$ and $b \in B$. That is: $A \times B = \{(a, b) \mid (a \in A) \wedge (b \in B)\}$. The Cartesian product of the sets $A_1, A_2, \ldots, A_n$, denoted by $A_1 \times A_2 \times \cdots \times A_n$, is the set of ordered n-tuples $(a_1, a_2, ..., a_n)$, where a_i belongs to A_i for $i = 1, 2, \ldots, n$. That is,

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) \mid a_i \in A_i \text{ for } i = 1, 2, \ldots, n\}$$

Rna mutation insertion deletion example

Trace the pseudocode to find the output of *mutation*($(\texttt{AUC}, 3, \texttt{G})$)

Fill in the blanks so that *insertion*($(\texttt{AUC}, \_, \_)$) $= \texttt{AUCG}$

Fill in the blanks so that *deletion*($(\_\_, \_)$) $= \texttt{G}$

Rna rnalen basecount definitions

Recall the definitions: The set of RNA strands S is defined (recursively) by:

$$\begin{array}{ll} \text{Basis Step:} & \mathtt{A} \in S, \mathtt{C} \in S, \mathtt{U} \in S, \mathtt{G} \in S \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B \text{, then } sb \in S \end{array}$$

where sb is string concatenation.

The function *rnalen* that computes the length of RNA strands in S is defined recursively by:

$$\begin{array}{lllll} & & rnalen : S & \to \mathbb{Z}^+ \\ \text{Basis Step:} & \text{If } b \in B \text{ then} & rnalen(b) & = 1 \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B \text{, then} & rnalen(sb) & = 1 + rnalen(s) \end{array}$$

The function *basecount* that computes the number of a given base b appearing in a RNA strand s is defined recursively by:

$$\begin{array}{llll} & & basecount : S \times B & \to \mathbb{N} \\ \text{Basis Step:} & \text{If } b_1 \in B, b_2 \in B & basecount(\ (b_1, b_2)\ ) & = \begin{cases} 1 & \text{when } b_1 = b_2 \\ 0 & \text{when } b_1 \neq b_2 \end{cases} \\ \text{Recursive Step:} & \text{If } s \in S, b_1 \in B, b_2 \in B & basecount(\ (sb_1, b_2)\ ) & = \begin{cases} 1 + basecount(\ (s, b_2)\ ) & \text{when } b_1 = b_2 \\ basecount(\ (s, b_2)\ ) & \text{when } b_1 \neq b_2 \end{cases} \end{array}$$

Proof strategies quantification finite domain

When a predicate $P(x)$ is over a **finite** domain:

- To show that $\forall x P(x)$ is true: check that $P(x)$ evaluates to T at each domain element by evaluating over and over. This is called "Proof of universal by **exhaustion**".
- To show that $\forall x P(x)$ is false: find a **counterexample**, a domain element where $P(x)$ evaluates to F.
- To show that $\exists x P(x)$ is true: find a **witness**, a domain element where $P(x)$ evaluates to T.
- To show that $\exists x P(x)$ is false: check that $P(x)$ evaluates to F at each domain element by evaluating over and over. DeMorgan's Law gives that $\neg \exists x P(x) \equiv \forall x \neg P(x)$ so this amounts to a proof of universal by exhaustion.

Proof strategy universal generalization

New! Proof by universal generalization: To prove that $\forall x\, P(x)$ is true, we can take an arbitrary element e from the domain of quantification and show that $P(e)$ is true, without making any assumptions about e other than that it comes from the domain.
An **arbitrary** element of a set or domain is a fixed but unknown element from that set.

Quiz translating counting quantifiers

Suppose $P(x)$ is a predicate over a domain D.

1. True or False: To translate the statement "There are at least two elements in D where the predicate P evaluates to true", we could write

$$\exists x_1 \in D\, \exists x_2 \in D\, (P(x_1) \wedge P(x_2))$$

2. True or False: To translate the statement "There are at most two elements in D where the predicate P evaluates to true", we could write

$$\forall x_1 \in D\, \forall x_2 \in D\, \forall x_3 \in D\ (\ (\ P(x_1) \wedge P(x_2) \wedge P(x_3)\ ) \rightarrow (\ x_1 = x_2 \vee x_2 = x_3 \vee x_1 = x_3\ )\ )$$

Sets equality subset definition

Definitions:

A **set** is an unordered collection of elements. When A and B are sets, $A = B$ (set equality) means

$$\forall x (x \in A \leftrightarrow x \in B)$$

When A and B are sets, $A \subseteq B$ ("A is a **subset** of B") means

$$\forall x (x \in A \rightarrow x \in B)$$

When A and B are sets, $A \subsetneq B$ ("A is a **proper subset** of B") means

$$(A \subseteq B) \wedge (A \neq B)$$

Proof strategies conditionals

New! Proof of conditional by direct proof: To prove that the conditional statement $p \to q$ is true, we can assume p is true and use that assumption to show q is true.

New! Proof of conditional by contrapositive proof: To prove that the implication $p \to q$ is true, we can assume q is false and use that assumption to show p is also false.

New! Proof of disjuction using equivalent conditional: To prove that the disjunction $p \vee q$ is true, we can rewrite it equivalently as $\neg p \to q$ and then use direct proof or contrapositive proof.

Proof strategies proof by cases

New! Proof by Cases: To prove q, we can work by cases by first describing all possible cases we might be in and then showing that each one guarantees q. Formally, if we know that $p_1 \vee p_2$ is true, and we can show that $(p_1 \to q)$ is true and we can show that $(p_2 \to q)$, then we can conclude q is true.

Proof strategies ands

New! Proof of conjunctions with subgoals: To show that $p \wedge q$ is true, we have two subgoals: subgoal (1) prove p is true; and, subgoal (2) prove q is true.

To show that $p \wedge q$ is false, it's enough to prove that $\neg p$.
To show that $p \wedge q$ is false, it's enough to prove that $\neg q$.

Sets proof strategies

To prove that one set is a subset of another, e.g. to show $A \subseteq B$:

To prove that two sets are equal, e.g. to show $A = B$:

Sets equality example

Example: $\{43, 7, 9\} = \{7, 43, 9, 7\}$

Sets basic proofs

Prove or **disprove**: $\{\texttt{A}, \texttt{C}, \texttt{U}, \texttt{G}\} \subseteq \{\texttt{AA}, \texttt{AC}, \texttt{AU}, \texttt{AG}\}$

Prove or **disprove**: For some set B, $\emptyset \in B$.

Prove or **disprove**: For every set B, $\emptyset \in B$.

Prove or **disprove**: The empty set is a subset of every set.

Prove or **disprove**: The empty set is a proper subset of every set.

Prove or **disprove**: $\{4, 6\} \subseteq \{n \mid \exists c \in \mathbb{Z}(n = 4c)\}$

Prove or **disprove**: $\{4, 6\} \subseteq \{n \textbf{ mod } 10 \mid \exists c \in \mathbb{Z}(n = 4c)\}$

Proofs signposting

Consider $\dots$, an **arbitrary** $\dots$. **Assume** $\dots$, we **want to show** that $\dots$. Which is what was needed, so the proof is complete $\square$.

or, in other words:

Let $\dots$ be an **arbitrary** $\dots$. **Assume** $\dots$, **WTS** that $\dots$ **QED**.

Set operations union intersection powerset

Cartesian product: When A and B are sets,

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$$

Example: $\{43, 9\} \times \{9, \mathbb{Z}\} =$

Example: $\mathbb{Z} \times \emptyset =$

Union: When A and B are sets,

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

Example: $\{43, 9\} \cup \{9, \mathbb{Z}\} =$

Example: $\mathbb{Z} \cup \emptyset =$

Intersection: When A and B are sets,

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

Example: $\{43, 9\} \cap \{9, \mathbb{Z}\} =$

Example: $\mathbb{Z} \cap \emptyset =$

Set difference: When A and B are sets,

$$A - B = \{x \mid x \in A \wedge x \notin B\}$$

Example: $\{43, 9\} - \{9, \mathbb{Z}\} =$

Example: $\mathbb{Z} - \emptyset =$

Disjoint sets: sets A and B are disjoint means $A \cap B = \emptyset$

Example: $\{43, 9\}, \{9, \mathbb{Z}\}$ are not disjoint

Example: The sets $\mathbb{Z}$ and $\emptyset$ are disjoint

Power set: When U is a set, $\mathcal{P}(U) = \{X \mid X \subseteq U\}$

Example: $\mathcal{P}(\{43, 9\}) =$

Example: $\mathcal{P}(\emptyset) =$

Logical operators full truth table

Input		Output				
		Conjunction	Exclusive or	Disjunction	Conditional	Biconditional
p	q	$p \wedge q$	$p \oplus q$	$p \vee q$	$p \to q$	$p \leftrightarrow q$
T	T	T	F	T	T	T
T	F	F	T	T	F	F
F	T	F	T	T	T	F
F	F	F	F	F	T	T
		"p and q"	"p xor q"	"p or q"	"if p then q"	"p if and only if q"

Hypothesis conclusion

The only way to make the conditional statement $p \to q$ false is to ________________________________

The **hypothesis** of $p \to q$ is __________ The **antecedent** of $p \to q$ is __________

The **conclusion** of $p \to q$ is __________ The **consequent** of $p \to q$ is __________

Converse inverse contrapositive

The **converse** of $p \to q$ is ____________________

The **inverse** of $p \to q$ is ____________________

The **contrapositive** of $p \to q$ is ____________________

Compound propositions recursive definition

We can use a recursive definition to describe all compound propositions that use propositional variables from a specified collection. Here's the definition for all compound propositions whose propositional variables are in $\{p, q\}$.

Basis Step:	p and q are each a compound proposition
Recursive Step:	If x is a compound proposition then so is $(\neg x)$ and if x and y are both compound propositions then so is each of $(x \wedge y), (x \oplus y), (x \vee y), (x \to y), (x \leftrightarrow y)$

Compound propositions precedence

Order of operations (Precedence) for logical operators:

Negation, then conjunction / disjunction, then conditional / biconditionals.

Example: $\neg p \vee \neg q$ means $(\neg p) \vee (\neg q)$.

Logical equivalence identities

(Some) logical equivalences

Can replace p and q with any compound proposition

$\neg(\neg p) \equiv p$		**Double negation**
$p \vee q \equiv q \vee p$	$p \wedge q \equiv q \wedge p$	**Commutativity** Ordering of terms
$(p \vee q) \vee r \equiv p \vee (q \vee r)$	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	**Associativity** Grouping of terms
$p \wedge F \equiv F$ $\quad$ $p \vee T \equiv T$	$p \wedge T \equiv p$ $\quad$ $p \vee F \equiv p$	**Domination** aka short circuit evaluation
$\neg(p \wedge q) \equiv \neg p \vee \neg q$	$\neg(p \vee q) \equiv \neg p \wedge \neg q$	**DeMorgan's Laws**

$p \to q \equiv \neg p \vee q$

$p \to q \equiv \neg q \to \neg p$ **Contrapositive**

$\neg(p \to q) \equiv p \wedge \neg q$

$\neg(p \leftrightarrow q) \equiv p \oplus q$

$p \leftrightarrow q \equiv q \leftrightarrow p$

Extra examples:

$p \leftrightarrow q$ is not logically equivalent to $p \wedge q$ because ____________________________________

$p \to q$ is not logically equivalent to $q \to p$ because ____________________________________

Logical operators english synonyms

Common ways to express logical operators in English:

Negation $\neg p$ can be said in English as

- Not p.
- It's not the case that p.
- p is false.

Conjunction $p \wedge q$ can be said in English as

- p and q.
- Both p and q are true.
- p but q.

Exclusive or $p \oplus q$ can be said in English as

- p or q, but not both.
- Exactly one of p and q is true.

Disjunction $p \vee q$ can be said in English as

- p or q, or both.
- p or q (inclusive).
- At least one of p and q is true.

Conditional $p \to q$ can be said in English as

- if p, then q.
- p is sufficient for q.
- q when p.
- q whenever p.
- p implies q.
- q follows from p.
- p is sufficient for q.
- q is necessary for p.
- p only if q.

Biconditional

- p if and only if q.
- p iff q.
- If p then q, and conversely.
- p is necessary and sufficient for q.

Compound propositions translation

Translation: Express each of the following sentences as compound propositions, using the given propositions.

"A sufficient condition for the warranty to be good is that you bought the computer less than a year ago"

w is "the warranty is good"
b is "you bought the computer less than a year ago"

"Whenever the message was sent from an unknown system, it is scanned for viruses."

s is "The message is scanned for viruses"
u is "The message was sent from an unknown system"

"I will complete my to-do list only if I put a reminder in my calendar"

d is "I will complete my to-do list"
c is "I put a reminder in my calendar"

Consistency def

Definition: A collection of compound propositions is called **consistent** if there is an assignment of truth values to the propositional variables that makes each of the compound propositions true.

Predicate definition

Definition: A **predicate** is a function from a given set (domain) to $\{T, F\}$.

A predicate can be applied, or **evaluated** at, an element of the domain.

Usually, a predicate *describes a property* that domain elements may or may not have.

Two predicates over the same domain are **equivalent** means they evaluate to the same truth values for all possible assignments of domain elements to the input. In other words, they are equivalent means that they are equal as functions.

To define a predicate, we must specify its domain and its value at each domain element. The rule assigning truth values to domain elements can be specified using a formula, English description, in a table (if the domain is finite), or recursively (if the domain is recursively defined).

Predicate examples finite domain

Input	Output		
	$V(x)$	$N(x)$	$Mystery(x)$
x	$[x]_{2c,3} > 0$	$[x]_{2c,3} < 0$	
000	F		T
001	T		T
010	T		T
011	T		F
100	F		F
101	F		T
110	F		F
111	F		T

The domain for each of the predicates $V(x)$, $N(x)$, $Mystery(x)$ is ______________________________.

Fill in the table of values for the predicate $N(x)$ based on the formula given.

Predicate truth set definition

Definition: The **truth set** of a predicate is the collection of all elements in its domain where the predicate evaluates to T.

Notice that specifying the domain and the truth set is sufficient for defining a predicate.

Predicate truth set example

The truth set for the predicate $V(x)$ is _________________________.

The truth set for the predicate $N(x)$ is _________________________.

The truth set for the predicate $Mystery(x)$ is ____________________________________.

Quantification definition

The **universal quantification** of predicate $P(x)$ over domain U is the statement "$P(x)$ for all values of x in the domain U" and is written $\forall x P(x)$ or $\forall x \in U\ P(x)$. When the domain is finite, universal quantification over the domain is equivalent to iterated *conjunction* (ands).

The **existential quantification** of predicate $P(x)$ over domain U is the statement "There exists an element x in the domain U such that $P(x)$" and is written $\exists x P(x)$ for $\exists x \in U\ P(x)$. When the domain is finite, existential quantification over the domain is equivalent to iterated *disjunction* (ors).

An element for which $P(x) = F$ is called a **counterexample** of $\forall x P(x)$.

An element for which $P(x) = T$ is called a **witness** of $\exists x P(x)$.

Quantification logical equivalence

Statements involving predicates and quantifiers are **logically equivalent** means they have the same truth value no matter which predicates (domains and functions) are substituted in.

Quantifier version of De Morgan's laws: $\boxed{\neg\forall x P(x) \ \equiv\ \exists x\,(\neg P(x))}$ $\boxed{\neg\exists x Q(x) \ \equiv\ \forall x\,(\neg Q(x))}$

Quantification examples finite domain

Examples of quantifications using $V(x), N(x), Mystery(x)$:

True or **False**: $\exists x\ (\ V(x) \wedge N(x)\ )$

True or **False**: $\forall x\ (\ V(x) \to N(x)\ )$

True or **False**: $\exists x\ (\ N(x) \leftrightarrow Mystery(x)\ )$

Rewrite $\neg\forall x\ (\ V(x) \oplus Mystery(x)\ )$ into a logical equivalent statement.

Notice that these are examples where the predicates have *finite* domain. How would we evaluate quantifications where the domain may be infinite?

Rna rnalen basecount definitions

Recall the definitions: The set of RNA strands S is defined (recursively) by:

$$\begin{array}{ll} \text{Basis Step:} & \mathtt{A} \in S, \mathtt{C} \in S, \mathtt{U} \in S, \mathtt{G} \in S \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B \text{, then } sb \in S \end{array}$$

where sb is string concatenation.

The function *rnalen* that computes the length of RNA strands in S is defined recursively by:

$$\begin{array}{llll} & & rnalen : S & \to \mathbb{Z}^+ \\ \text{Basis Step:} & \text{If } b \in B \text{ then} & rnalen(b) & = 1 \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B \text{, then} & rnalen(sb) & = 1 + rnalen(s) \end{array}$$

The function *basecount* that computes the number of a given base b appearing in a RNA strand s is defined recursively by:

$$\begin{array}{llll} & & basecount : S \times B & \to \mathbb{N} \\ \text{Basis Step:} & \text{If } b_1 \in B, b_2 \in B & basecount(\ (b_1, b_2)\ ) & = \begin{cases} 1 & \text{when } b_1 = b_2 \\ 0 & \text{when } b_1 \neq b_2 \end{cases} \\ \text{Recursive Step:} & \text{If } s \in S, b_1 \in B, b_2 \in B & basecount(\ (sb_1, b_2)\ ) & = \begin{cases} 1 + basecount(\ (s, b_2)\ ) & \text{when } b_1 = b_2 \\ basecount(\ (s, b_2)\ ) & \text{when } b_1 \neq b_2 \end{cases} \end{array}$$

Predicate rna example

Example predicates on S, the set of RNA strands (an infinite set)

$H : S \to \{T, F\}$ where $H(s) = T$ for all s.

Truth set of H is ______________

$F_{\mathtt{A}} : S \to \{T, F\}$ defined recursively by:

Basis step: $F_{\mathtt{A}}(\mathtt{A}) = T$, $F_{\mathtt{A}}(\mathtt{C}) = F_{\mathtt{A}}(\mathtt{G}) = F_{\mathtt{A}}(\mathtt{U}) = F$

Recursive step: If $s \in S$ and $b \in B$, then $F_{\mathtt{A}}(sb) = F_{\mathtt{A}}(s)$.

Example where $F_{\mathtt{A}}$ evaluates to T is ________________

Example where $F_{\mathtt{A}}$ evaluates to F is ________________

Predicates example rnalen basecount

Using functions to define predicates:

L with domain $S \times \mathbb{Z}^+$ is defined by, for $s \in S$ and $n \in \mathbb{Z}^+$,

$$L(\ (s,n)\ ) = \begin{cases} T & \text{if } rnalen(s) = n \\ F & \text{otherwise} \end{cases}$$

In other words, $L(\ (s,n)\ )$ means $rnalen(s) = n$

BC with domain $S \times B \times \mathbb{N}$ is defined by, for $s \in S$ and $b \in B$ and $n \in \mathbb{N}$,

$$BC(\ (s,b,n)\ ) = \begin{cases} T & \text{if } basecount(\ (s,b)\ ) = n \\ F & \text{otherwise} \end{cases}$$

In other words, $BC(\ (s,b,n)\ )$ means $basecount(\ (s,b)\ ) = n$

Example where L evaluates to T: ________________ Why?

Example where BC evaluates to T: ________________ Why?

Example where L evaluates to F: ________________ Why?

Example where BC evaluates to F: ________________ Why?

$\exists t\ BC(t)$ $\qquad$ $\exists (s,b,n) \in S \times B \times \mathbb{N}\ (basecount(\ (s,b)\ ) = n)$

In English:

Witness that proves this existential quantification is true:

$\forall t\ BC(t)$ $\qquad$ $\forall (s,b,n) \in S \times B \times \mathbb{N}\ (basecount(\ (s,b)\ ) = n)$

In English:

Counterexample that proves this universal quantification is false:

Predicates projecting example rna basecount

New predicates from old

1. Define the **new** predicate with domain $S \times B$ and rule

$$basecount(\ (s, b)\ ) = 3$$

Example domain element where predicate is T:

2. Define the **new** predicate with domain $S \times \mathbb{N}$ and rule

$$basecount(\ (s, \mathtt{A})\ ) = n$$

Example domain element where predicate is T:

3. Define the **new** predicate with domain $S \times B$ and rule

$$\exists n \in \mathbb{N}\ (basecount(\ (s, b)\ ) = n)$$

Example domain element where predicate is T:

4. Define the **new** predicate with domain S and rule

$$\forall b \in B\ (basecount(\ (s, b)\ ) = 1)$$

Example domain element where predicate is T:

Predicate notation

Notation: for a predicate P with domain $X_1 \times \cdots \times X_n$ and a n-tuple $(x_1, \ldots, x_n)$ with each $x_i \in X$, we can write $P(x_1, \ldots, x_n)$ to mean $P(\ (x_1, \ldots, x_n)\ )$.

Nested quantifiers

Nested quantifiers

$$\forall s \in S \; \forall b \in B \; \forall n \in \mathbb{N} \; (basecount(\ (s,b)\ ) = n)$$

In English:

Counterexample that proves this universal quantification is false:

$$\forall n \in \mathbb{N} \; \forall s \in S \; \forall b \in B \; (basecount(\ (s,b)\ ) = n)$$

In English:

Counterexample that proves this universal quantification is false:

Sets proof strategies

To prove that one set is a subset of another, e.g. to show $A \subseteq B$:

To prove that two sets are equal, e.g. to show $A = B$:

Sets basic proofs operations

Let $W = \mathcal{P}(\{1, 2, 3, 4, 5\})$

Example elements in W are:

Prove or **disprove**: $\forall A \in W \, \forall B \in W \; (A \subseteq B \; \rightarrow \; \mathcal{P}(A) \subseteq \mathcal{P}(B))$

Prove or **disprove**: $\forall A \in W \, \forall B \in W \; (\mathcal{P}(A) = \mathcal{P}(B) \; \rightarrow \; A = B)$

Prove or **disprove**: $\forall A \in W \, \forall B \in W \, \forall C \in W \; (A \cup B = A \cup C \; \rightarrow \; B = C)$

Proof strategies road map

We now have propositional and predicate logic that can help us express statements about any domain. We will develop proof strategies to craft valid argument for proving that such statements are true or disproving them (by showing they are false). We will practice these strategies with statements about sets and numbers, both because they are familiar and because they can be used to build cryptographic systems. Then we will apply proof strategies more broadly to prove statements about data structures and machine learning applications.

Numbers facts

1. Addition and multiplication of real numbers are each commutative and associative.

2. The product of two positive numbers is positive, of two negative numbers is positive, and of a positive and a negative number is negative.

3. The sum of two integers, the product of two integers, and the difference between two integers are each integers.

4. For every integer x there is no integer strictly between x and $x + 1$,

5. When x, y are positive integers, $xy \geq x$ and $xy \geq y$.

Factoring definition

Definition: When a and b are integers and a is nonzero, a **divides** b means there is an integer c such that $b = ac$.

Symbolically, $F(\ (a, b)\ ) =$ and is a predicate over the domain ______

Other (synonymous) ways to say that $F(\ (a, b)\ )$ is true:

a is a **factor** of b $\quad$ a is a **divisor** of b $\quad$ b is a **multiple** of a $\quad$ $a|b$

When a is a positive integer and b is any integer, $a|b$ exactly when b **mod** $a = 0$

When a is a positive integer and b is any integer, $a|b$ exactly $b = a \cdot (b \textbf{ div } a)$

Factoring translation examples

Translate these quantified statements by matching to English statement on right.

$\exists a \in \mathbb{Z}^{\neq 0}\ (\ F(\ (a, a)\ )\ )$	Every nonzero integer is a factor of itself.
$\exists a \in \mathbb{Z}^{\neq 0}\ (\ \neg F(\ (a, a)\ )\ )$	No nonzero integer is a factor of itself.
$\forall a \in \mathbb{Z}^{\neq 0}\ (\ F(\ (a, a)\ )\ )$	At least one nonzero integer is a factor of itself.
$\forall a \in \mathbb{Z}^{\neq 0}\ (\ \neg F(\ (a, a)\ )\ )$	Some nonzero integer is not a factor of itself.

Factoring basic claims

Claim: Every nonzero integer is a factor of itself.

Proof:

Prove or **Disprove**: There is a nonzero integer that does not divide its square.

Prove or **Disprove**: Every positive factor of a positive integer is less than or equal to it.

Factoring basic claims continued

Claim: Every nonzero integer is a factor of itself and every nonzero integer divides its square.

Factoring even odd

Definition: an integer n is **even** means that there is an integer a such that $n = 2a$; an integer n is **odd** means that there is an integer a such that $n = 2a + 1$. Equivalently, an integer n is **even** means $n \mathbf{\ mod\ } 2 = 0$; an integer n is **odd** means $n \mathbf{\ mod\ } 2 = 1$. Also, an integer is even if and only if it is not odd.

Prime number definition

Definition: An integer p greater than 1 is called **prime** means the only positive factors of p are 1 and p. A positive integer that is greater than 1 and is not prime is called composite.

Primes basic claims

Extra examples: Use the definition to prove that 1 is not prime, 2 is prime, 3 is prime, 4 is not prime, 5 is prime, 6 is not prime, and 7 is prime.

True or False: The statement "There are three consecutive positive integers that are prime."

Hint: These numbers would be of the form $p, p+1, p+2$ (where p is a positive integer).

Proof: We need to show ________________________________

True or False: The statement "There are three consecutive odd positive integers that are prime."

Hint: These numbers would be of the form $p, p+2, p+4$ (where p is an odd positive integer).

Proof: We need to show ____

Rna rnalen basecount definitions

Recall the definitions: The set of RNA strands S is defined (recursively) by:

$$\begin{array}{ll} \text{Basis Step:} & \mathtt{A} \in S, \mathtt{C} \in S, \mathtt{U} \in S, \mathtt{G} \in S \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B \text{, then } sb \in S \end{array}$$

where sb is string concatenation.

The function *rnalen* that computes the length of RNA strands in S is defined recursively by:

$$\begin{array}{llll} & & rnalen : S & \to \mathbb{Z}^+ \\ \text{Basis Step:} & \text{If } b \in B \text{ then} & rnalen(b) & = 1 \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B \text{, then} & rnalen(sb) & = 1 + rnalen(s) \end{array}$$

The function *basecount* that computes the number of a given base b appearing in a RNA strand s is defined recursively by:

$$\begin{array}{llll} & & basecount : S \times B & \to \mathbb{N} \\ \text{Basis Step:} & \text{If } b_1 \in B, b_2 \in B & basecount(\ (b_1, b_2)\ ) & = \begin{cases} 1 & \text{when } b_1 = b_2 \\ 0 & \text{when } b_1 \neq b_2 \end{cases} \\ \text{Recursive Step:} & \text{If } s \in S, b_1 \in B, b_2 \in B & basecount(\ (sb_1, b_2)\ ) & = \begin{cases} 1 + basecount(\ (s, b_2)\ ) & \text{when } b_1 = b_2 \\ basecount(\ (s, b_2)\ ) & \text{when } b_1 \neq b_2 \end{cases} \end{array}$$

Structural induction motivating example rna

Claim $\forall s \in S \; (\; rnalen(s) > 0 \;)$

Proof: Let s be an arbitrary RNA strand. By the recursive definition of S, either $s \in B$ or there is some strand s_0 and some base b such that $s = s_0 b$. We will show that the inequality holds for both cases.

Case: Assume $s \in B$. We need to show $rnalen(s) > 0$. By the basis step in the definition of $rnalen$,

$$rnalen(s) = 1$$

which is greater than 0, as required.

Case: Assume there is some strand s_0 and some base b such that $s = s_0 b$. We will show *(the stronger claim)* that

$$\forall u \in S \; \forall b \in B \; (\; rnalen(u) > 0 \to rnalen(ub) > 0 \;)$$

Consider an arbitrary RNA strand u and an arbitrary base b, and assume towards a direct proof, that

$$rnalen(u) > 0$$

We need to show that $rnalen(ub) > 0$.

$$rnalen(ub) = 1 + rnalen(u) > 1 + 0 = 1 > 0$$

as required.

Proof strategies structural induction

Proof by Structural Induction To prove a universal quantification over a recursively defined set:

Basis Step: Show the statement holds for elements specified in the basis step of the definition.

Recursive Step: Show that if the statement is true for each of the elements used to construct new elements in the recursive step of the definition, the result holds for these new elements.

Structural induction example rnalen basecount

Claim $\forall s \in S\, (\mathit{rnalen(s)} \geq \mathit{basecount}(\ (s, \texttt{A})\ ))$:

Proof: We proceed by structural induction on the recursively defined set S.

Basis Case: We need to prove that the inequality holds for each element in the basis step of the recursive definition of S. Need to show

$$(\mathit{rnalen}(\texttt{A}) \geq \mathit{basecount}(\ (\texttt{A}, \texttt{A})\ )\ ) \wedge (\ \mathit{rnalen}(\texttt{C}) \geq \mathit{basecount}(\ (\texttt{C}, \texttt{A})\ )\ )$$
$$\wedge(\ \mathit{rnalen}(\texttt{U}) \geq \mathit{basecount}(\ (\texttt{U}, \texttt{A})\ )\ ) \wedge (\ \mathit{rnalen}(\texttt{G}) \geq \mathit{basecount}(\ (\texttt{G}, \texttt{A})\ )\ )$$

We calculate, using the definitions of *rnalen* and *basecount*:

Recursive Case: We will prove that

$$\forall u \in S\ \forall b \in B\ (\ \mathit{rnalen}(u) \geq \mathit{basecount}(\ (u, \texttt{A})\ ) \to \mathit{rnalen}(ub) \geq \mathit{basecount}(\ (ub, \texttt{A})\ )$$

Consider arbitrary RNA strand u and arbitrary base b. Assume, as the **induction hypothesis**, that $\mathit{rnalen}(u) \geq \mathit{basecount}(\ (u, \texttt{A})\ )$. We need to show that $\mathit{rnalen}(ub) \geq \mathit{basecount}(\ (ub, \texttt{A})\ )$.

Using the recursive step in the definition of the function *rnalen*:

$$\mathit{rnalen}(ub) = 1 + \mathit{rnalen}(u)$$

The recursive step in the definition of the function *basecount* has two cases. We notice that $b = \texttt{A} \vee b \neq \texttt{A}$ and we proceed by cases.

Case i. Assume $b = \texttt{A}$.

Using the first case in the recursive step in the definition of the function *basecount*:

$$\mathit{basecount}(\ (ub, \texttt{A})\ ) = 1 + \mathit{basecount}(\ (u, \texttt{A})\ )$$

By the **induction hypothesis**, we know that $\mathit{basecount}(\ (u, \texttt{A})\ ) \leq \mathit{rnalen}(u)$ so:

$$\mathit{basecount}(\ (ub, \texttt{A})\ ) = 1 + \mathit{basecount}(\ (u, \texttt{A})\ ) \leq 1 + \mathit{rnalen}(u) = \mathit{rnalen}(ub)$$

and, thus, $\mathit{basecount}(\ (ub, \texttt{A})\ ) \leq \mathit{rnalen}(ub)$, as required.

Case ii. Assume $b \neq \texttt{A}$.

Using the second case in the recursive step in the definition of the function *basecount*:

$$\mathit{basecount}(\ (ub, \texttt{A})\ ) = \mathit{basecount}(\ (u, \texttt{A})\ )$$

By the **induction hypothesis**, we know that $basecount(\ (u, \texttt{A})\ ) \leq rnalen(u)$ so:

$$basecount(\ (ub, \texttt{A})\ ) = basecount(\ (u, \texttt{A})\ ) \leq rnalen(u) < 1 + rnalen(u) = rnalen(ub)$$

and, thus, $basecount(\ (ub, \texttt{A})\ ) \leq rnalen(ub)$, as required.

Proofs signposting kinds of claims

To organize our proofs, it's useful to highlight which claims are most important for our overall goals. We use some terminology to describe different roles statements can have.

Theorem: Statement that can be shown to be true, usually an important one.

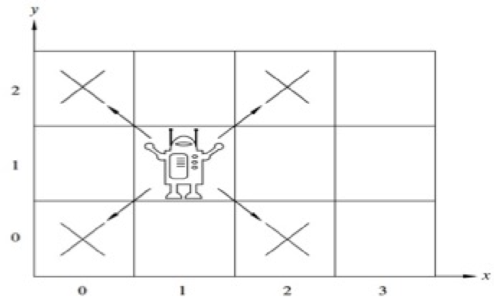
Less important theorems can be called **proposition**, **fact**, **result**, **claim**.

Lemma: A less important theorem that is useful in proving a theorem.

Corollary: A theorem that can be proved directly after another one has been proved, without needing a lot of extra work.

Invariant: A theorem that describes a property that is true about an algorithm or system no matter what inputs are used.

Structural induction example robot grid



Theorem: A robot on an infinite 2-dimensional integer grid starts at $(0,0)$ and at each step moves to diagonally adjacent grid point. This robot can / cannot (*circle one*) reach $(1,0)$.

Definition The set of positions the robot can visit Pos is defined by:

Basis Step: $(0,0) \in Pos$
Recursive Step: If $(x,y) \in Pos$, then

are also in Pos

Example elements of Pos are:

Lemma: $\forall (x,y) \in Pos \ \ (x+y \text{ is an even integer })$

Why are we calling this a lemma?

Proof of theorem using lemma: To show is $(1,0) \notin Pos$. Rewriting the lemma to explicitly restrict the domain of the universal, we have $\forall (x,y)$ ($(x,y) \in Pos \ \ \rightarrow \ \ (x+y$ is an even integer)). Since the universal is true, ($(1,0) \in Pos \ \ \rightarrow \ \ (1+0$ is an even integer)) is a true statement. Evaluating the conclusion of this conditional statement: By definition of long division, since $1 = 0 \cdot 2 + 1$ (where $0 \in \mathbb{Z}$ and $1 \in \mathbb{Z}$ and $0 \leq 1 < 2$ mean that 0 is the quotient and 1 is the remainder), $1 \textbf{ mod } 2 = 1$ which is not 0 so the conclusion is false. A true conditional with a false conclusion must have a false hypothesis: $(1,0) \notin Pos$, QED. $\square$

Proof of lemma by structural induction:

Basis Step:

Recursive Step: Consider arbitrary $(x,y) \in Pos$. To show is:

$$(x+y \text{ is an even integer}) \rightarrow (\text{sum of coordinates of next position is even integer})$$

Assume **as the induction hypothesis, IH** that:

Structural induction example sum of powers

The set $\mathbb{N}$ is recursively defined. Therefore, the function $sumPow : \mathbb{N} \to \mathbb{N}$ which computes, for input i, the sum of the nonnegative powers of 2 up to and including exponent i is defined recursively by

$$\begin{array}{ll} \text{Basis step:} & sumPow(0) = 1 \\ \text{Recursive step:} & \text{If } x \in \mathbb{N}, \text{ then} \quad sumPow(x+1) = sumPow(x) + 2^{x+1} \end{array}$$

$sumPow(0) =$

$sumPow(1) =$

$sumPow(2) =$

Fill in the blanks in the following proof of

$$\forall n \in \mathbb{N}\ (sumPow(n) = 2^{n+1} - 1)$$

Proof: Since $\mathbb{N}$ is recursively defined, we proceed by ____________________.

Basis case: We need to show that ____________________. Evaluating each side: $LHS = sumPow(0) = 1$ by the basis case in the recursive definition of $sumPow$; $RHS = 2^{0+1} - 1 = 2^1 - 1 = 2 - 1 = 1$. Since $1 = 1$, the equality holds.

Recursive case: Consider arbitrary natural number n and assume, as the ____________________ that $sumPow(n) = 2^{n+1} - 1$. We need to show that ____________________. Evaluating each side:

$$LHS = sumPow(n+1) \overset{\text{rec def}}{=} sumPow(n) + 2^{n+1} \overset{\text{IH}}{=} (2^{n+1} - 1) + 2^{n+1}.$$

$$RHS = 2^{(n+1)+1} - 1 \overset{\text{exponent rules}}{=} 2 \cdot 2^{n+1} - 1 = \left(2^{n+1} + 2^{n+1}\right) - 1 \overset{\text{regrouping}}{=} (2^{n+1} - 1) + 2^{n+1}$$

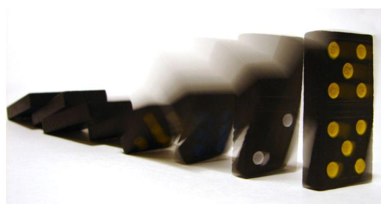
Thus, $LHS = RHS$. The structural induction is complete and we have proved the universal generalization. $\square$

Proof strategy mathematical induction

Proof by Mathematical Induction
To prove a universal quantification over the set of all integers greater than or equal to some base integer b,

> **Basis Step**: Show the property holds for b.
>
> **Recursive Step**: Consider an arbitrary integer n greater than or equal to b, assume (as the **induction hypothesis**) that the property holds for n, and use this and other facts to prove that the property holds for $n+1$.

Induction dominos



Wikimedia commons

https://creativecommons.org/licenses/by/2.0/legalcode

Proof strategy mathematical induction

Proof by Mathematical Induction
To prove a universal quantification over the set of all integers greater than or equal to some base integer b,

> **Basis Step**: Show the property holds for b.
>
> **Recursive Step**: Consider an arbitrary integer n greater than or equal to b, assume (as the **induction hypothesis**) that the property holds for n, and use this and other facts to prove that the property holds for $n+1$.

Proof strategy strong induction

Proof by Strong Induction
To prove that a universal quantification over the set of all integers greater than or equal to some base integer b holds, pick a fixed nonnegative integer j and then:

Basis Step:	Show the statement holds for b, $b+1$, $\ldots$, $b+j$.
Recursive Step:	Consider an arbitrary integer n greater than or equal to $b+j$, assume (as the **strong induction hypothesis**) that the property holds for **each of** b, $b+1$, $\ldots$, n, and use this and other facts to prove that the property holds for $n+1$.

Binary expansions exist proof

Theorem: Every positive integer is a sum of (one or more) distinct powers of 2. *Binary expansions exist!*

Recall the definition for binary expansion:

Definition For n a positive integer, the **binary expansion of** n is

$$(a_{k-1} \cdots a_1 a_0)_b$$

where k is a positive integer, $a_0, a_1, \ldots, a_{k-1}$ are each 0 or 1, $a_{k-1} \neq 0$, and

$$n = \sum_{i=0}^{k-1} a_i b^i$$

The idea in the "Least significant first" algorithm for computing binary expansions is that the binary expansion of *half* a number becomes *part* of the binary expansion of the number of itself. We can use this idea in a proof by strong induction that binary expansions exist for all positive integers n.

Proof by strong induction, with $b = 1$ and $j = 0$.

Basis step: WTS property is true about 1.

Recursive step: Consider an arbitrary integer $n \geq 1$.

Assume (as the strong induction hypothesis, IH) that the property is true about each of $1, \ldots, n$.

WTS that the property is true about $n + 1$.

Idea: We will apply the IH to $(n + 1)$ **div** 2.

Why is this ok?

Why is this helpful?

By the IH, we can write $(n+1)$ **div** 2 as a sum of powers of 2. In other words, there are values $a_{k-1}, \ldots, a_0$ such that each a_i is 0 or 1, $a_{k-1} = 1$, and

$$\sum_{i=0}^{k-1} a_i 2^i = (n+1) \textbf{ div } 2$$

Define the collection of coefficients

$$c_j = \begin{cases} a_{j-1} & \text{if } 1 \leq j \leq k \\ (n+1) \textbf{ mod } 2 & \text{if } j = 0 \end{cases}$$

Calculating:

$$\begin{aligned}
\sum_{j=0}^{k} c_j 2^j = c_0 + \sum_{j=1}^{k} c_j 2^j = c_0 + \sum_{i=0}^{k-1} c_{i+1} 2^{i+1} & & \text{re-indexing the summation} \\
= c_0 + 2 \cdot \sum_{i=0}^{k-1} c_{i+1} 2^i & & \text{factoring out a 2 from each term in the sum} \\
= c_0 + 2 \cdot \sum_{i=0}^{k-1} a_i 2^i & & \text{by definition of } c_{i+1} \\
= c_0 + 2 \, (\ (n+1) \textbf{ div } 2\ ) & & \text{by IH} \\
= (\ (n+1) \textbf{ mod } 2\ ) + 2 \, (\ (n+1) \textbf{ div } 2\ ) & & \text{by definition of } c_0 \\
= n+1 & & \text{by definition of long division}
\end{aligned}$$

Thus, $n+1$ can be expressed as a sum of powers of 2, as required.

Fundamental theorem proof

Theorem: Every positive integer *greater than 1* is a product of (one or more) primes.

Before we prove, let's try some examples:

$20 =$

$100 =$

$5 =$

Proof by strong induction, with $b = 2$ and $j = 0$.

Basis step: WTS property is true about 2.

Since 2 is itself prime, it is already written as a product of (one) prime.

Recursive step: Consider an arbitrary integer $n \geq 2$. Assume (as the strong induction hypothesis, IH) that the property is true about each of $2, \ldots, n$. WTS that the property is true about $n + 1$: We want to show that $n + 1$ can be written as a product of primes. Notice that $n + 1$ is itself prime or it is composite.

Case 1: assume $n + 1$ is prime and then immediately it is written as a product of (one) prime so we are done.

Case 2: assume that $n + 1$ is composite so there are integers x and y where $n + 1 = xy$ and each of them is between 2 and n (inclusive). Therefore, the induction hypothesis applies to each of x and y so each of these factors of $n + 1$ can be written as a product of primes. Multiplying these products together, we get a product of primes that gives $n + 1$, as required.

Since both cases give the necessary conclusion, the proof by cases for the recursive step is complete.

Strong induction making change proof idea

Suppose we had postage stamps worth 5 cents and 3 cents. Which number of cents can we form using these stamps? In other words, which postage can we pay?

11?

15?

4?

$$\begin{aligned}
&CanPay(0) \land \neg CanPay(1) \land \neg CanPay(2) \land \\
&CanPay(3) \land \neg CanPay(4) \land CanPay(5) \land CanPay(6) \\
&\neg CanPay(7) \land \forall n \in \mathbb{Z}^{\geq 8} CanPay(n)
\end{aligned}$$

where the predicate $CanPay$ with domain $\mathbb{N}$ is

$$CanPay(n) = \exists x \in \mathbb{N} \exists y \in \mathbb{N} (5x + 3y = n)$$

Proof (idea): First, explicitly give witnesses or general arguments for postages between 0 and 7. To prove the universal claim, we can use mathematical induction or strong induction.

Approach 1, mathematical induction: if we have stamps that add up to n cents, need to use them (and others) to give $n + 1$ cents. How do we get 1 cent with just 3-cent and 5-cent stamps?
Either take away a 5-cent stamps and add two 3-cent stamps,
or take away three 3-cent stamps and add two 5-cent stamps.
The details of this proof by mathematical induction are making sure we have enough stamps to use one of these approaches.

Approach 2, strong induction: assuming we know how to make postage for **all** smaller values (greater than or equal to 8), when we need to make $n+1$ cents, add one 3 cent stamp to however we make $(n + 1) - 3$ cents.
The details of this proof by strong induction are making sure we stay in the domain of the universal when applying the induction hypothesis.

Strong induction nim

Consider the following game: two players start with two (equal) piles of jellybeans in front of them. They take turns removing any positive integer number of jellybeans at a time from one of two piles in front of them in turns.

The player who removes the last jellybean wins the game.

Which player (if any) has a strategy to guarantee to win the game?

Try out some games, starting with 1 jellybean in each pile, then 2 jellybeans in each pile, then 3 jellybeans in each pile. Who wins in each game?

Notice that reasoning about the strategy for the 1 jellybean game is easier than about the strategy for the 2 jellybean game.

Formulate a winning strategy by working to transform the game to a simpler one we know we can win.

Player 2's Strategy: Take the same number of jellybeans that Player 1 did, but from the opposite pile.

Why is this a good idea: If Player 2 plays this strategy, at the next turn Player 1 faces a game with the same setup as the original, just with fewer jellybeans in the two piles. Then Player 2 can keep playing this strategy to win.

Claim: Player 2's strategy guarantees they will win the game.

Proof: By strong induction, we will prove that for all positive integers n, Player 2's strategy guarantees a win in the game that starts with n jellybeans in each pile.

Basis step: WTS Player 2's strategy guarantees a win when each pile starts with 1 jellybean.

In this case, Player 1 has to take the jellybean from one of the piles (because they can't take from both piles at once). Following the strategy, Player 2 takes the jellybean from the other pile, and wins because this is the last jellybean.

Recursive step: Let n be a positive integer. As the strong induction hypothesis, assume that Player 2's strategy guarantees a win in the games where there are $1, 2, \ldots, n$ many jellybeans in each pile at the start of the game.

WTS that Player 2's strategy guarantees a win in the game where there are $n + 1$ in the jellybeans in each pile at the start of the game.

In this game, the first move has Player 1 take some number, call it c (where $1 \leq c \leq n + 1$), of jellybeans from one of the piles. Playing according to their strategy, Player 2 then takes the same number of jellybeans from the other pile.

Notice that $(c = n + 1) \vee (c \leq n)$.

Case 1: Assume $c = n + 1$, then in their first move, Player 2 wins because they take all of the second pile, which includes the last jellybean.

Case 2: Assume $c \leq n$. Then after Player 2's first move, the two piles have an equal number of jellybeans. The number of jellybeans in each pile is

$$(n + 1) - c$$

and, since $1 \leq c \leq n$, this number is between 1 and n. Thus, at this stage of the game, the game appears identical to a new game where the two piles have an equal number of jellybeans between 1 and n. Thus, the strong induction hypothesis applies, and Player 2's strategy guarantees they win.

Linked lists definition

Definition The set of linked lists of natural numbers L is defined recursively by

Basis Step:	$[] \in L$
Recursive Step:	If $l \in L$ and $n \in \mathbb{N}$, then $(n, l) \in L$

Linked lists examples

Visually:

Example: the list with two nodes whose first node has 20 and whose second node has 42

Linked list length definition

Definition: The length of a linked list of natural numbers L, $length : L \to \mathbb{N}$ is defined by

Basis Step:		$length(\ []\ )$	$= 0$
Recursive Step:	If $l \in L$ and $n \in \mathbb{N}$, then	$length(\ (n, l)\ )$	$= 1 + length(l)$

Linked lists prepend definition

Definition: The function $prepend : L \times \mathbb{N} \to L$ that adds an element at the front of a linked list is defined by

Linked list append definition

Definition The function $append : L \times \mathbb{N} \to L$ that adds an element at the end of a linked list is defined by

Basis Step:	If $m \in \mathbb{N}$ then
Recursive Step:	If $l \in L$ and $n \in \mathbb{N}$ and $m \in \mathbb{N}$, then

Linked list append length claim proof

Claim: $\forall l \in L$ ($length($ $append($ $(l, 100)$)) $> length(l)$)

Proof: By structural induction on L, we have two cases:

Basis Step

1. **To Show** $length(\ append(\ ([], 100)\ )\ ) > length(\ []\ )$	Because $[]$ is the only element defined in the basis step of L, we only need to prove that the property holds for $[]$.
2. **To Show** $length(\ (100, [])\ ) > length(\ []\ )$	By basis step in definition of *append*.
3. **To Show** $(1 + length(\ []\ )) > length(\ []\ )$	By recursive step in definition of *length*.
4. **To Show** $1 + 0 > 0$	By basis step in definition of *length*.
5. T	By properties of integers
QED	Because we got to T only by rewriting **To Show** to equivalent statements, using well-defined proof techniques, and applying definitions.

Recursive Step

Consider an arbitrary: $l' \in L$, $n \in \mathbb{N}$, and we assume as the **induction hypothesis** that:

$$length(\ append(\ (l', 100\ )\ )\ ) > length(l')$$

Our goal is to show that $length(\ append(\ (\ (n, l'), 100\ )\ )\ ) > length(\ (n, l')\ )$ is also true. We start by working with one side of the candidate inequality:

$$\begin{aligned} LHS &= length(\ append(\ (\ (n, l'), 100\ )\ )\ ) \\ &= length(\ (n, append(\ (l', 100)\ )\ )\ ) \quad \text{by the recursive definition of } append \\ &= 1 + length(\ append(\ (l', 100)\ )\ ) \quad \text{by the recursive definition of } length \\ &> 1 + length(l') \quad \text{by the induction hypothesis} \\ &= length((n, l')) \quad \text{by the recursive definition of } length \\ &= RHS \end{aligned}$$

Linked list example each length

Prove or disprove: $\forall n \in \mathbb{N}\ \exists l \in L\ (\ length(l) = n\ )$

Proof strategy proof by contradiction

New! Proof by Contradiction
To prove that a statement p is true, pick another statement r and once we show that $\neg p \to (r \wedge \neg r)$ then we can conclude that p is true.
Informally The statement we care about can't possibly be false, so it must be true.

Least greatest proofs

For a set of numbers X, how do you formalize “there is a greatest X” or “there is a least X”?

Prove or **disprove**: There is a least prime number.

Prove or **disprove**: There is a greatest integer.

Approach 1, De Morgan’s and universal generalization:

Approach 2, proof by contradiction:

Extra examples: Prove or disprove that $\mathbb{N}$, $\mathbb{Q}$ each have a least and a greatest element.

Gcd definition

Definition: **Greatest common divisor** Let a and b be integers, not both zero. The largest integer d such that d is a factor of a and d is a factor of b is called the greatest common divisor of a and b and is denoted by $gcd(\ (a,b)\ )$.

Gcd examples

Why do we restrict to the situation where a and b are not both zero?

Calculate $gcd(\ (10,15)\ )$

Calculate $gcd(\ (10,20)\ )$

Gcd basic claims

Claim: For any integers a, b (not both zero), $gcd(\ (a,b)\ ) \geq 1$.

Proof: *Show that 1 is a common factor of any two integers, so since the gcd is the greatest common factor it is greater than or equal to any common factor.*

Claim: For any positive integers a, b, $gcd(\ (a,b)\ ) \leq a$ and $gcd(\ (a,b)\ ) \leq b$.

Proof *Using the definition of gcd and the fact that factors of a positive integer are less than or equal to that integer.*

Claim: For any positive integers a, b, if a divides b then $gcd(\ (a,b)\ ) = a$.

Proof *Using previous claim and definition of gcd.*

Claim: For any positive integers a, b, c, if there is some integer q such that $a = bq + c$,

$$gcd(\ (a,b)\ ) = gcd(\ (b,c)\ )$$

Proof *Prove that any common divisor of* a, b *divides* c *and that any common divisor of* b, c *divides* a*.*

Gcd lemma relatively prime

Lemma: For any integers p, q (not both zero), $gcd\left(\left(\frac{p}{gcd(\ (p,q)\ )}, \frac{q}{gcd(\ (p,q)\ )} \right) \right) = 1$. In other words, can reduce to relatively prime integers by dividing by gcd.

Proof:

Let x be arbitrary positive integer and assume that x is a factor of each of $\frac{p}{gcd(\ (p,q)\ )}$ and $\frac{q}{gcd(\ (p,q)\ )}$. This gives integers α, β such that

$$\alpha x = \frac{p}{gcd(\ (p,q)\ )} \qquad \beta x = \frac{q}{gcd(\ (p,q)\ )}$$

Multiplying both sides by the denominator in the RHS:

$$\alpha x \cdot gcd(\ (p,q)\ ) = p \qquad \beta x \cdot gcd(\ (p,q)\ ) = q$$

In other words, $x \cdot gcd(\ (p,q)\ )$ is a common divisor of p, q. By definition of gcd, this means

$$x \cdot gcd(\ (p,q)\ ) \leq gcd(\ (p,q)\ )$$

and since $gcd(\ (p,q)\ )$ is positive, this means, $x \leq 1$.

Rational numbers definition

The **set of rational numbers**, $\mathbb{Q}$ is defined as

$$\left\{\frac{p}{q} \mid p \in \mathbb{Z} \text{ and } q \in \mathbb{Z} \text{ and } q \neq 0\right\} \quad \text{or, equivalently,} \quad \{x \in \mathbb{R} \mid \exists p \in \mathbb{Z} \exists q \in \mathbb{Z}^+ (p = x \cdot q)\}$$

Extra practice: Use the definition of set equality to prove that the definitions above give the same set.

Proof by contradiction irrational

Goal: The square root of 2 is not a rational number. In other words: $\neg \exists x \in \mathbb{Q}(x^2 - 2 = 0)$

Attempted proof: The definition of the set of rational numbers is the collection of fractions p/q where p is an integer and q is a nonzero integer. Looking for a **witness** p and q, we can write the square root of 2 as the fraction $\sqrt{2}/1$, where 1 is a nonzero integer. Since the numerator is not in the domain, this witness is not allowed, and we have shown that the square root of 2 is not a fraction of integers (with nonzero denominator). Thus, the square root of 2 is not rational.

The problem in the above attempted proof is that ____

Lemma 1: For every two integers a and b, not both zero, with $gcd(\ (a,b)\ ) = 1$, it is not the case that both a is even and b is even.

Lemma 2: For every integer x, x is even if and only if x^2 is even.

Proof: Towards a proof by contradiction, we will define a statement r such that $\sqrt{2} \in \mathbb{Q} \to (r \wedge \neg r)$.

Assume that $\sqrt{2} \in \mathbb{Q}$. Namely, there are positive integers p, q such that

$$\sqrt{2} = \frac{p}{q}$$

Let $a = \frac{p}{gcd(\ (p,q)\ )}$, $b = \frac{q}{gcd(\ (p,q)\ )}$, then

$$\sqrt{2} = \frac{a}{b} \qquad \text{and} \qquad gcd(\ (a,b)\ ) = 1$$

By Lemma 1, a and b are not both even. We define r to be the statement "a is even and b is even", and we have proved $\neg r$.

Squaring both sides and clearing denominator: $2b^2 = a^2$.

By definition of even, since b^2 is an integer, a^2 is even.

By Lemma 2, this guarantees that a is even too. So, by definition of even, there is some integer (call it c), such that $a = 2c$.

Plugging into the equation:

$$2b^2 = a^2 = (2c)^2 = 4c^2$$

and dividing both sides by 2

$$b^2 = 2c^2$$

and since c^2 is an integer, b^2 is even. By Lemma 2, b is even too. Thus, a is even and b is even and we have proved r.

In other words, assuming that $\sqrt{2} \in \mathbb{Q}$ guarantees $r \wedge \neg r$, which is impossible, so $\sqrt{2} \notin \mathbb{Q}$. QED

Compound proposition definitions

Proposition: Declarative sentence that is true or false (not both).

Propositional variable: Variable that represents a proposition.

Compound proposition: New proposition formed from existing propositions (potentially) using logical operators. *Note*: A propositional variable is one example of a compound proposition.

Truth table: Table with one row for each of the possible combinations of truth values of the input and an additional column that shows the truth value of the result of the operation corresponding to a particular row.

Logical equivalence

Logical equivalence : Two compound propositions are **logically equivalent** means that they have the same truth values for all settings of truth values to their propositional variables.

Tautology: A compound proposition that evaluates to true for all settings of truth values to its propositional variables; it is abbreviated T.

Contradiction: A compound proposition that evaluates to false for all settings of truth values to its propositional variables; it is abbreviated F.

Contingency: A compound proposition that is neither a tautology nor a contradiction.

Tautology contradiction contingency examples

Label each of the following as a tautology, contradiction, or contingency.

$p \wedge p$

$p \oplus p$

$p \vee p$

$p \vee \neg p$

$p \wedge \neg p$

Logical equivalence extra example

Extra Example: Which of the compound propositions in the table below are logically equivalent?

Input		Output				
p	q	$\neg(p \wedge \neg q)$	$\neg(\neg p \vee \neg q)$	$(\neg p \vee q)$	$(\neg q \vee \neg p)$	$(p \wedge q)$
T	T					
T	F					
F	T					
F	F					

Algorithm definition

New! An algorithm is a finite sequence of precise instructions for solving a problem.

Algorithms can be expressed in English or in more formalized descriptions like pseudocode or fully executable programs.

Sometimes, we can define algorithms whose output matches the rule for a function we already care about. Consider the (integer) logarithm function

$$logb : \{b \in \mathbb{Z} \mid b > 1\} \times \mathbb{Z}^+ \quad \to \quad \mathbb{N}$$

defined by

$$logb(\ (b, n)\ ) = \text{greatest integer } y \text{ so that } b^y \text{ is less than or equal to } n$$

Calculating integer part of base b logarithm

```
procedure logb(b,n: positive integers with b>1)
i := 0
while n > b-1
  i := i+1
  n := n div b
return i {i holds the integer part of the base b logarithm of n}
```

Trace this algorithm with inputs $b = 3$ and $n = 17$

	b	n	i	$n > b - 1$?
Initial value	3	17		
After 1 iteration				
After 2 iterations				
After 3 iterations				

Compare: does the output match the rule for the (integer) logarithm function?

Fixed width definition

Definition For b an integer greater than 1, w a positive integer, and n a nonnegative integer ____________,
the **base b fixed-width w expansion of n** is

$$(a_{w-1} \cdots a_1 a_0)_{b,w}$$

where $a_0, a_1, \ldots, a_{w-1}$ are nonnegative integers less than b and

$$n = \sum_{i=0}^{w-1} a_i b^i$$

Fixed width example

Decimal $b = 10$	Binary $b = 2$	Binary fixed-width 10 $b = 2$, $w = 10$	Binary fixed-width 7 $b = 2$, $w = 7$	Binary fixed-width 4 $b = 2$, $w = 4$
$(20)_{10}$				

Fixed width fractional definition

Definition For b an integer greater than 1, w a positive integer, w' a positive integer, and x a real number the **base b fixed-width expansion of x with integer part width w and fractional part width w'** is $(a_{w-1} \cdots a_1 a_0 . c_1 \cdots c_{w'})_{b,w,w'}$ where $a_0, a_1, \ldots, a_{w-1}, c_1, \ldots, c_{w'}$ are nonnegative integers less than b and

$$x \geq \sum_{i=0}^{w-1} a_i b^i + \sum_{j=1}^{w'} c_j b^{-j} \qquad \text{and} \qquad x < \sum_{i=0}^{w-1} a_i b^i + \sum_{j=1}^{w'} c_j b^{-j} + b^{-w'}$$

3.75 in fixed-width binary, integer part width 2, fractional part width 8	
0.1 in fixed-width binary, integer part width 2, fractional part width 8	

```
welcome $jshell
|  Welcome to JShell -- Version 10.0.1
|  For an introduction type: /help intro

jshell> 0.1
$1 ==>

jshell> 0.2
$2 ==>

jshell> 0.1 + 0.2
$3 ==>

jshell> Math.sqrt(2)
$4 ==>

jshell> Math.sqrt(2)*Math.sqrt(2)
$5 ==>

jshell> 
```

Note: Java uses floating point, not fixed width representation, but similar rounding errors appear in both.

Negative int expansions

Representing negative integers in binary: Fix a positive integer width for the representation w, $w > 1$.

	To represent a positive integer n	To represent a negative integer $-n$
Sign-magnitude	$[0a_{w-2} \cdots a_0]_{s,w}$, where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $n = 17$, $w = 7$:	$[1a_{w-2} \cdots a_0]_{s,w}$, where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $-n = -17$, $w = 7$:
2s complement	$[0a_{w-2} \cdots a_0]_{2c,w}$, where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $n = 17$, $w = 7$:	$[1a_{w-2} \cdots a_0]_{2c,w}$, where $2^{w-1} - n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $-n = -17$, $w = 7$:

Calculating 2s complement

For positive integer n, to represent $-n$ in 2s complement with width w,

- Calculate $2^{w-1} - n$, convert result to binary fixed-width $w - 1$, pad with leading 1, or
- Express $-n$ as a sum of powers of 2, where the leftmost 2^{w-1} is negative weight, or
- Convert n to binary fixed-width w, flip bits, add 1 (ignore overflow)

Challenge: use definitions to explain why each of these approaches works.

Representing zero

Representing 0:

So far, we have representations for positive and negative integers. What about 0?

	To represent a **non-negative** integer n	To represent a **non-positive** integer $-n$
Sign-magnitude	$[0a_{w-2} \cdots a_0]_{s,w}$, where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $n = 0$, $w = 7$:	$[1a_{w-2} \cdots a_0]_{s,w}$, where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $-n = 0$, $w = 7$:
2s complement	$[0a_{w-2} \cdots a_0]_{2c,w}$, where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $n = 0$, $w = 7$:	$[1a_{w-2} \cdots a_0]_{2c,w}$, where $2^{w-1} - n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $-n = 0$, $w = 7$:

Netflix intro

What data should we encode about each Netflix account holder to help us make effective recommendations?

In machine learning, clustering can be used to group similar data for prediction and recommendation. For example, each Netflix user's viewing history can be represented as a n-tuple indicating their preferences about movies in the database, where n is the number of movies in the database. People with similar tastes in movies can then be clustered to provide recommendations of movies for one another. Mathematically, clustering is based on a notion of distance between pairs of n-tuples.

Data types

Term	**Examples**: (add additional examples from class)
set unordered collection of elements *repetition doesn't matter* *Equal sets agree on membership of all elements*	$7 \in \{43, 7, 9\}$ $\quad$ $2 \notin \{43, 7, 9\}$
n**-tuple** ordered sequence of elements with n "slots" ($n > 0$) *repetition matters, fixed length* *Equal n-tuples have corresponding components equal*	
string ordered finite sequence of elements each from specified set (called the alphabet over which the string is defined) *repetition matters, arbitrary finite length* *Equal strings have same length and corresponding characters equal*	

Special cases:

When $n = 2$, the 2-tuple is called an **ordered pair**.

A string of length 0 is called the **empty string** and is denoted λ.

A set with no elements is called the **empty set** and is denoted $\{\}$ or $\emptyset$.

Set operations

To define a set we can use the roster method, set builder notation, a recursive definition, and also we can apply a set operation to other sets.

New! Cartesian product of sets and **set-wise concatenation of sets of strings**

Definition: Let X and Y be sets. The **Cartesian product** of X and Y, denoted $X \times Y$, is the set of all ordered pairs (x, y) where $x \in X$ and $y \in Y$

$$X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$$

Conventions: (1) Cartesian products can be chained together to result in sets of n-tuples and (2) When we form the Cartesian product of a set with itself $X \times X$ we can denote that set as X^2, or X^n for the Cartesian product of a set with itself n times for a positive integer n.

Definition: Let X and Y be sets of strings over the same alphabet. The **set-wise concatenation** of X and Y, denoted $X \circ Y$, is the set of all results of string concatenation xy where $x \in X$ and $y \in Y$

$$X \circ Y = \{xy \mid x \in X \text{ and } y \in Y\}$$

Pro-tip: the meaning of writing one element next to another like xy depends on the data-types of x and y. When x and y are strings, the convention is that xy is the result of string concatenation. When x and y are numbers, the convention is that xy is the result of multiplication. This is (one of the many reasons) why is it very important to declare the data-type of variables before we use them.

Fill in the missing entries in the table:

Set	Example elements in this set and their data type:
B	$\texttt{A} \quad \texttt{C} \quad \texttt{G} \quad \texttt{U}$
	$(\texttt{A}, \texttt{C}) \qquad (\texttt{U}, \texttt{U})$
$B \times \{-1, 0, 1\}$	
$\{-1, 0, 1\} \times B$	
	$(0, 0, 0)$
$\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\} \circ \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$	
	GGGG

Defining functions

New! Defining functions A function is defined by its (1) domain, (2) codomain, and (3) rule assigning each element in the domain exactly one element in the codomain.

The domain and codomain are nonempty sets.
The rule can be depicted as a table, formula, piecewise definition, or English description.
The notation is

"Let the function FUNCTION-NAME: DOMAIN $\to$ CODOMAIN be given by
FUNCTION-NAME(x) = ... for every $x \in DOMAIN$".

or

"Consider the function FUNCTION-NAME: DOMAIN $\to$ CODOMAIN defined as
FUNCTION-NAME(x) = ... for every $x \in DOMAIN$".

Example: The absolute value function

Domain

Codomain

Rule

Defining functions recursively

When the domain of a function is a *recursively defined set*, the rule assigning images to domain elements (outputs) can also be defined recursively.

Recall: The set of RNA strands S is defined (recursively) by:

$$\begin{array}{ll} \text{Basis Step:} & \texttt{A} \in S, \texttt{C} \in S, \texttt{U} \in S, \texttt{G} \in S \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B \text{, then } sb \in S \end{array}$$

where sb is string concatenation.

Definition (Of a function, recursively) A function *rnalen* that computes the length of RNA strands in S is defined by:

$$\begin{array}{lllll} & & \mathit{rnalen} : S & \to \mathbb{Z}^+ \\ \text{Basis Step:} & \text{If } b \in B \text{ then} & \mathit{rnalen}(b) & = 1 \\ \text{Recursive Step:} & \text{If } s \in S \text{ and } b \in B \text{, then} & \mathit{rnalen}(sb) & = 1 + \mathit{rnalen}(s) \end{array}$$

The domain of *rnalen* is

The codomain of *rnalen* is

Example function application:

$$\mathit{rnalen}(\texttt{ACU}) =$$

Example: A function *basecount* that computes the number of a given base b appearing in a RNA strand s is defined recursively:

$$\begin{array}{llll} & & \mathit{basecount} : S \times B & \to \mathbb{N} \\ \text{Basis Step:} & \text{If } b_1 \in B, b_2 \in B & \mathit{basecount}(\ (b_1, b_2)\ ) & = \begin{cases} 1 & \text{when } b_1 = b_2 \\ 0 & \text{when } b_1 \neq b_2 \end{cases} \\ \text{Recursive Step:} & \text{If } s \in S, b_1 \in B, b_2 \in B & \mathit{basecount}(\ (sb_1, b_2)\ ) & = \begin{cases} 1 + \mathit{basecount}(\ (s, b_2)\ ) & \text{when } b_1 = b_2 \\ \mathit{basecount}(\ (s, b_2)\ ) & \text{when } b_1 \neq b_2 \end{cases} \end{array}$$