

## 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  $A_1, A_2, \dots, A_n$ , denoted by  $A_1 \times A_2 \times \dots \times A_n$ , is the set of ordered n-tuples  $(a_1, a_2, \dots, a_n)$ , where  $a_i$  belongs to  $A_i$  for  $i = 1, 2, \dots, n$ . That is,

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

## Rna mutation insertion deletion example

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

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

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

# Rna rna len basecount definitions

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

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

where  $sb$  is string concatenation.

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

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

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

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

## Alternating quantifiers order rna examples

### Alternating nested quantifiers

$$\forall s \in S \exists n \in \mathbb{N} ( basecount( (s, \mathbf{U}) ) = n )$$

In English: For each strand, there is a nonnegative integer that counts the number of occurrences of  $\mathbf{U}$  in that strand.

$$\exists n \in \mathbb{N} \forall s \in S ( basecount( (s, \mathbf{U}) ) = n )$$

In English: There is a nonnegative integer that counts the number of occurrences of  $\mathbf{U}$  in every strand.

Are these statements true or false?

$$\forall s \in S \exists b \in B ( \text{basecount}(s, b) = 3 )$$

In English: For each RNA strand there is a base that occurs 3 times in this strand.

Write the negation and use De Morgan's law to find a logically equivalent version where the negation is applied only to the  $BC$  predicate (not next to a quantifier).

Is the original statement **True** or **False**?

## 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

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

## Proof strategies conditionals

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

**New! Proof of disjunction using equivalent conditional:** To prove that the disjunction  $p \vee q$  is true, we can rewrite it equivalently as  $\neg p \rightarrow q$  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  $(p_1 \rightarrow q)$  is true and we can show that  $(p_2 \rightarrow q)$ , 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:**  $\{A, C, U, G\} \subseteq \{AA, AC, AU, AG\}$

**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:**  $\{4, 6\} \subseteq \{n \bmod 10 \mid \exists c \in \mathbb{Z}(n = 4c)\}$

# Proofs signposting

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

*or, in other words:*

Let ... be an **arbitrary** .... **Assume** ..., **WTS** that ... **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:  $\mathcal{P}(\emptyset) =$

# 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:**  $\neg \forall x P(x) \equiv \exists x (\neg P(x))$

$\neg \exists x Q(x) \equiv \forall x (\neg Q(x))$

## 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:**  $\forall x ( V(x) \rightarrow N(x) )$

**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 rna len basecount definitions

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

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

where  $sb$  is string concatenation.

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

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

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

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

# 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

$$\text{basecount}( (s, b) ) = 3$$

Example domain element where predicate is  $T$ :

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

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

Example domain element where predicate is  $T$ :

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

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

Example domain element where predicate is  $T$ :

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

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

Example domain element where predicate is  $T$ :

# Nested quantifiers

## Nested quantifiers

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

In English:

Counterexample that proves this universal quantification is false:

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

In English:

Counterexample that proves this universal quantification is false:

# Alternating quantifiers strategies rna examples

## Alternating nested quantifiers

$$\forall s \in S \exists b \in B ( \text{basecount}(s, b) = 3 )$$

In English: For each RNA strand there is a base that occurs 3 times in this strand.

Write the negation and use De Morgan's law to find a logically equivalent version where the negation is applied only to the  $BC$  predicate (not next to a quantifier).

Is the original statement **True** or **False**?

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

In English: There is an RNA strand so that for each base there is some nonnegative integer that counts the number of occurrences of that base in this strand.

Write the negation and use De Morgan's law to find a logically equivalent version where the negation is applied only to the  $BC$  predicate (not next to a quantifier).

Is the original statement **True** or **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  $b$        $a$  is a **divisor** of  $b$        $b$  is a **multiple** of  $a$        $a|b$

When  $a$  is a positive integer and  $b$  is any integer,  $a|b$  exactly when  $b \bmod a = 0$

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

## 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  $n \bmod 2 = 0$ ; an integer  $n$  is **odd** means  $n \bmod 2 = 1$ . 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 rna len basecount definitions

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

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

where  $sb$  is string concatenation.

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

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

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

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

# Alternating quantifiers proofs rna examples

Which proof strategies could be used to prove each of the following statements?

*Hint: first translate the statements to English and identify the main logical structure.*

$$\forall s \in S \ ( \text{rnalen}(s) > 0 )$$

$$\forall b \in B \ \exists s \in S \ ( \text{basecount}( \ (s, b) \ ) > 0 )$$

$$\forall s \in S \ \exists b \in B \ ( \text{basecount}( \ (s, b) \ ) > 0 )$$

$$\exists s \in S \ ( \text{rnalen}(s) = \text{basecount}( \ (s, \mathbf{A}) \ ) )$$

$$\forall s \in S \ ( \text{rnalen}(s) \geq \text{basecount}( \ (s, \mathbf{A}) \ ) )$$

# Structural induction motivating example rna

**Claim**  $\forall s \in S ( \text{rnalen}(s) > 0 )$

**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  $s = s_0b$ . We will show that the inequality holds for both cases.

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

$$\text{rnalen}(s) = 1$$

which is greater than 0, as required.

**Case:** Assume there is some strand  $s_0$  and some base  $b$  such that  $s = s_0b$ . We will show (*the stronger claim*) that

$$\forall u \in S \forall b \in B ( \text{rnalen}(u) > 0 \rightarrow \text{rnalen}(ub) > 0 )$$

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

$$\text{rnalen}(u) > 0$$

We need to show that  $\text{rnalen}(ub) > 0$ .

$$\text{rnalen}(ub) = 1 + \text{rnalen}(u) > 1 + 0 = 1 > 0$$

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**  $\forall s \in S (rnalen(s) \geq basecount( (s, \mathbf{A}) ))$ :

**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

$$\begin{aligned} & ( rnalen(\mathbf{A}) \geq basecount( (\mathbf{A}, \mathbf{A}) ) ) \wedge ( rnalen(\mathbf{C}) \geq basecount( (\mathbf{C}, \mathbf{A}) ) ) \\ & \wedge ( rnalen(\mathbf{U}) \geq basecount( (\mathbf{U}, \mathbf{A}) ) ) \wedge ( rnalen(\mathbf{G}) \geq basecount( (\mathbf{G}, \mathbf{A}) ) ) \end{aligned}$$

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

**Recursive Case:** We will prove that

$$\forall u \in S \forall b \in B ( rnalen(u) \geq basecount( (u, \mathbf{A}) ) \rightarrow rnalen(ub) \geq basecount( (ub, \mathbf{A}) ) )$$

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

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

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

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

*Case i.* Assume  $b = \mathbf{A}$ .

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

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

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

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

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

*Case ii.* Assume  $b \neq \mathbf{A}$ .

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

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



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

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

and, thus,  $basecount( (ub, \mathbf{A}) ) \leq rnalen(ub)$ , 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.

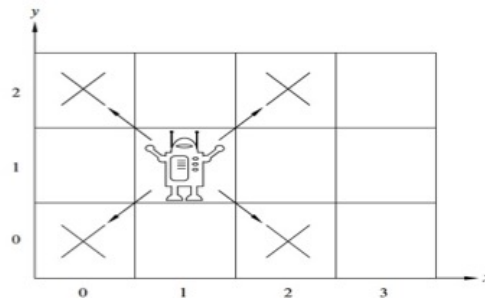
Less important theorems can be called **proposition**, **fact**, **result**, **claim**.

**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  $\forall (x,y) \ ( (x,y) \in Pos \rightarrow (x+y \text{ is an even integer}) )$ . Since the universal is true,  $( (1,0) \in Pos \rightarrow (1+0 \text{ is an even integer}) )$  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),  $1 \bmod 2 = 1$  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  $sumPow : \mathbb{N} \rightarrow \mathbb{N}$  which computes, for input  $i$ , the sum of the nonnegative powers of 2 up to and including exponent  $i$  is defined recursively by

Basis step:  $sumPow(0) = 1$

Recursive step: If  $x \in \mathbb{N}$ , then  $sumPow(x + 1) = sumPow(x) + 2^{x+1}$

$sumPow(0) =$

$sumPow(1) =$

$sumPow(2) =$

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:

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

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

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  $n + 1$ .

# Tautology contradiction contingency examples

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

$$p \wedge p$$

$$p \oplus p$$

$$p \vee p$$

$$p \vee \neg p$$

$$p \wedge \neg p$$

# Why represent numbers

Modeling uses data-types that are encoded in a computer. The details of the encoding impact the efficiency of algorithms we use to understand the systems we are modeling and the impacts of these algorithms on the people using the systems. Case study: how to encode numbers?

# 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  $a_0, a_1, \dots, a_{w-1}$  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  $a_0, a_1, \dots, a_{w-1}, c_1, \dots, c_{w'}$  are nonnegative integers less than  $b$  and

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

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 <b>non-negative</b> integer $n$	To represent a <b>non-positive</b> 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 $-n = 0, w = 7$ :