

## 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.

# 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  $y \in Y$

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

**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:			
$B$	A	C	G	U
	(A, C)		(U, U)	
$B \times \{-1, 0, 1\}$				
$\{-1, 0, 1\} \times B$				
	(0, 0, 0)			
$\{A, C, G, U\} \circ \{A, C, G, U\}$				
	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, or English description.

The notation is

“Let the function  $\text{FUNCTION-NAME: DOMAIN} \rightarrow \text{CODOMAIN}$  be given by  
 $\text{FUNCTION-NAME}(x) = \dots$  for every  $x \in \text{DOMAIN}$ ”.

or

“Consider the function  $\text{FUNCTION-NAME: DOMAIN} \rightarrow \text{CODOMAIN}$  given by  
 $\text{FUNCTION-NAME}(x) = \dots$  for every  $x \in \text{DOMAIN}$ ”.

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:

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

where  $sb$  is string concatenation.

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

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

The domain of  $rnalen$  is

The codomain of  $rnalen$  is

Example function application:

$$rnalen(\mathbf{ACU}) =$$

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

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

$$basecount((\mathbf{ACU}, \mathbf{A})) = basecount((\mathbf{AC}, \mathbf{A})) = basecount((\mathbf{A}, \mathbf{A})) = 1$$

$$basecount((\mathbf{ACU}, \mathbf{G})) = basecount((\mathbf{AC}, \mathbf{G})) = basecount((\mathbf{A}, \mathbf{G})) = 0$$

*Extra example:* The function which outputs  $2^n$  when given a nonnegative integer  $n$  can be defined recursively, because its domain is the set of nonnegative integers.

# 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?

## Base expansion definition

**Definition** For  $b$  an integer greater than 1 and  $n$  a positive integer, the **base  $b$  expansion of  $n$**  is

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

where  $k$  is a positive integer,  $a_0, a_1, \dots, a_{k-1}$  are nonnegative integers less than  $b$ ,  $a_{k-1} \neq 0$ , and

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

Notice: *The base  $b$  expansion of a positive integer  $n$  is a string over the alphabet  $\{x \in \mathbb{N} \mid x < b\}$  whose leftmost character is nonzero.*

Base $b$	Collection of possible coefficients in base $b$ expansion of a positive integer
Binary ( $b = 2$ )	$\{0, 1\}$
Ternary ( $b = 3$ )	$\{0, 1, 2\}$
Octal ( $b = 8$ )	$\{0, 1, 2, 3, 4, 5, 6, 7\}$
Decimal ( $b = 10$ )	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Hexadecimal ( $b = 16$ )	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ letter coefficient symbols represent numerical values $(A)_{16} = (10)_{10}$ $(B)_{16} = (11)_{10}$ $(C)_{16} = (12)_{10}$ $(D)_{16} = (13)_{10}$ $(E)_{16} = (14)_{10}$ $(F)_{16} = (15)_{10}$

# Base expansion examples

Common bases:	Binary $b = 2$	Octal $b = 8$	Decimal $b = 10$	Hexadecimal $b = 16$
---------------	----------------	---------------	------------------	----------------------

*Examples:*

$(1401)_2$

$(1401)_{10}$

$(1401)_{16}$

## Algorithm definition

<b>New!</b> An algorithm is a finite sequence of precise instructions for solving a problem.
--

## Algorithm half

Algorithm for calculating integer part of half the input

```
1  procedure half( $n$ : a positive integer)
2     $r := 0$ 
3    while  $n > 1$ 
4       $r := r + 1$ 
5       $n := n - 2$ 
6    return  $r$  { $r$  holds the result of the operation}
```

$n$	$r$	$n > 1?$
6		

$n$	$r$	$n > 1?$
5		

## Algorithm log

Algorithm for calculating integer part of log

```
1  procedure log( $n$ : a positive integer)
2     $r := 0$ 
3    while  $n > 1$ 
4       $r := r + 1$ 
5       $n := \text{half}(n)$ 
6    return  $r$  { $r$  holds the result of the log operation}
```

$n$	$r$	$n > 1?$
8		

$n$	$r$	$n > 1?$
6		

$2^0 = 1$	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$	$2^5 = 32$	$2^6 = 64$	$2^7 = 128$	$2^8 = 256$	$2^9 = 512$	$2^{10} = 1024$
-----------	-----------	-----------	-----------	------------	------------	------------	-------------	-------------	-------------	-----------------

## Division algorithm

**Integer division and remainders** (aka The Division Algorithm) Let  $n$  be an integer and  $d$  a positive integer. There are unique integers  $q$  and  $r$ , with  $0 \leq r < d$ , such that  $n = dq + r$ . In this case,  $d$  is called the divisor,  $n$  is called the dividend,  $q$  is called the quotient, and  $r$  is called the remainder. We write  $q = n \text{ div } d$  and  $r = n \text{ mod } d$ .

*Extra example:* How do **div** and **mod** compare to  $/$  and  $\%$  in Java and python?

## Base expansion algorithms

**Two algorithms for constructing base  $b$  expansion from decimal representation**

**Most significant first:** Start with left-most coefficient of expansion

Calculating integer part of  $\log_b$

```

1 procedure logb( $n, b$ : positive integers with  $b > 1$ )
2    $r := 0$ 
3   while  $n > b - 1$ 
4      $r := r + 1$ 
5      $n := n \text{ div } b$ 
6   return  $r$  { $r$  holds the result of the  $\log_b$  operation}
```

Calculating base  $b$  expansion, from left

```

1 procedure baseb1( $n, b$ : positive integers with  $b > 1$ )
2    $v := n$ 
3    $k := \log_b(n, b) + 1$ 
4   for  $i := 1$  to  $k$ 
5      $a_{k-i} := 0$ 
6     while  $v \geq b^{k-i}$ 
7        $a_{k-i} := a_{k-i} + 1$ 
8        $v := v - b^{k-i}$ 
9   return  $(a_{k-1}, \dots, a_0)\{(a_{k-1} \dots a_0)_b \text{ is the base } b \text{ expansion of } n\}$ 
```

**Least significant first:** Start with right-most coefficient of expansion

Idea: (when  $k > 1$ )

$$n = a_{k-1}b^{k-1} + \dots + a_1b + a_0$$

$$= b(a_{k-1}b^{k-2} + \dots + a_1) + a_0$$

so  $a_0 = n \text{ mod } b$  and  $a_{k-1}b^{k-2} + \dots + a_1 =$   
 $n \text{ div } b$ .

Calculating base  $b$  expansion, from right

```

1 procedure baseb2( $n, b$ : positive integers with  $b > 1$ )
2    $q := n$ 
3    $k := 0$ 
4   while  $q \neq 0$ 
5      $a_k := q \text{ mod } b$ 
6      $q := q \text{ div } b$ 
7      $k := k + 1$ 
8   return  $(a_{k-1}, \dots, a_0)\{(a_{k-1} \dots a_0)_b \text{ is the base } b \text{ expansion of } n\}$ 
```

## Base expansion review

Find and fix any and all mistakes with the following:

(a)  $(1)_2 = (1)_8$

(b)  $(142)_{10} = (142)_{16}$

(c)  $(20)_{10} = (10100)_2$

(d)  $(35)_8 = (1D)_{16}$



# Base conversion algorithm

Recall the definition of base expansion we discussed:

**Definition** For  $b$  an integer greater than 1 and  $n$  a positive integer, the **base  $b$  expansion of  $n$**  is

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

where  $k$  is a positive integer,  $a_0, a_1, \dots, a_{k-1}$  are nonnegative integers less than  $b$ ,  $a_{k-1} \neq 0$ , and

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

Notice: The base  $b$  expansion of a positive integer  $n$  is a string over the alphabet  $\{x \in \mathbb{N} \mid x < b\}$  whose leftmost character is nonzero.

Base $b$	Collection of possible coefficients in base $b$ expansion of a positive integer
Binary ( $b = 2$ )	$\{0, 1\}$
Ternary ( $b = 3$ )	$\{0, 1, 2\}$
Octal ( $b = 8$ )	$\{0, 1, 2, 3, 4, 5, 6, 7\}$
Decimal ( $b = 10$ )	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Hexadecimal ( $b = 16$ )	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ letter coefficient symbols represent numerical values $(A)_{16} = (10)_{10}$ $(B)_{16} = (11)_{10}$ $(C)_{16} = (12)_{10}$ $(D)_{16} = (13)_{10}$ $(E)_{16} = (14)_{10}$ $(F)_{16} = (15)_{10}$

We write an algorithm for converting from base  $b_1$  expansion to base  $b_2$  expansion:

# 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 $b = 2, w = 4$
$(20)_{10}$	(a)	(b)	(c)	(d)

# 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.

## Expansion summary

base $b$ expansion of $n$	base $b$ fixed-width $w$ expansion of $n$
For $b$ an integer greater than 1 and $n$ a positive integer, the <b>base <math>b</math> expansion of <math>n</math></b> is $(a_{k-1} \cdots a_1 a_0)_b$ where $k$ is a positive integer, $a_0, a_1, \dots, a_{k-1}$ are nonnegative integers less than $b$ , $a_{k-1} \neq 0$ , and $n = a_{k-1}b^{k-1} + \cdots + a_1b + a_0$	For $b$ an integer greater than 1, $w$ a positive integer, and $n$ a nonnegative integer with $n < b^w$ , the <b>base <math>b</math> fixed-width <math>w</math> expansion of <math>n</math></b> 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 $n = a_{w-1}b^{w-1} + \cdots + a_1b + a_0$

# 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 $-n = -17$ , $w = 7$ :
Extra example: 1s complement	$[0a_{w-2} \cdots a_0]_{1c,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $n = 17$ , $w = 7$ :	$[1\bar{a}_{w-2} \cdots \bar{a}_0]_{1c,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ and we define $\bar{0} = 1$ and $\bar{1} = 0$ . 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 $n = 0, w = 7$ :  (a)	$[1a_{w-2} \cdots a_0]_{s,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $-n = 0, w = 7$ :  (b)
2s complement	$[0a_{w-2} \cdots a_0]_{2c,w}$ , where $n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $n = 0, w = 7$ :  (c)	$[1a_{w-2} \cdots a_0]_{2c,w}$ , where $2^{w-1} - n = (a_{w-2} \cdots a_0)_{2,w-1}$ Example $-n = 0, w = 7$ :  (d)

# Fixed width addition

**Fixed-width addition:** adding one bit at time, using the usual column-by-column and carry arithmetic, and dropping the carry from the leftmost column so the result is the same width as the summands. *Does this give the right value for the sum?*

$$\begin{array}{r} (1\ 1\ 0\ 1\ 0\ 0)_{2,6} \\ + (0\ 0\ 0\ 1\ 0\ 1)_{2,6} \\ \hline \end{array}$$

$$\begin{array}{r} [1\ 1\ 0\ 1\ 0\ 0]_{s,6} \\ + [0\ 0\ 0\ 1\ 0\ 1]_{s,6} \\ \hline \end{array}$$

$$\begin{array}{r} [1\ 1\ 0\ 1\ 0\ 0]_{2c,6} \\ + [0\ 0\ 0\ 1\ 0\ 1]_{2c,6} \\ \hline \end{array}$$

## Circuits basics

In a **combinatorial circuit** (also known as a **logic circuit**), we have **logic gates** connected by **wires**. The inputs to the circuits are the values set on the input wires: possible values are 0 (low) or 1 (high). The values flow along the wires from left to right. A wire may be split into two or more wires, indicated with a filled-in circle (representing solder). Values stay the same along a wire. When one or more wires flow into a gate, the output value of that gate is computed from the input values based on the gate's definition table. Outputs of gates may become inputs to other gates.

## Logic gates definitions

Inputs		Output
$x$	$y$	$x$ AND $y$
1	1	1
1	0	0
0	1	0
0	0	0



Inputs		Output
$x$	$y$	$x$ XOR $y$
1	1	0
1	0	1
0	1	1
0	0	0



Input	Output
$x$	NOT $x$
1	0
0	1



# Digital circuits basic examples

Example digital circuit:



Output when  $x = 1, y = 0, z = 0, w = 1$  is \_\_\_\_\_

Output when  $x = 1, y = 1, z = 1, w = 1$  is \_\_\_\_\_

Output when  $x = 0, y = 0, z = 0, w = 1$  is \_\_\_\_\_

Draw a logic circuit with inputs  $x$  and  $y$  whose output is always 0. *Can you use exactly 1 gate?*

## Half adder circuit

**Fixed-width addition:** adding one bit at time, using the usual column-by-column and carry arithmetic, and dropping the carry from the leftmost column so the result is the same width as the summands. In many cases, this gives representation of the correct value for the sum when we interpret the summands in fixed-width binary or in 2s complement.

For single column:

Input		Output	
$x_0$	$y_0$	$c_0$	$s_0$
1	1		
1	0		
0	1		
0	0		



## Two bit adder circuit

Draw a logic circuit that implements binary addition of two numbers that are each represented in fixed-width binary:

- Inputs  $x_0, y_0, x_1, y_1$  represent  $(x_1x_0)_{2,2}$  and  $(y_1y_0)_{2,2}$
- Outputs  $z_0, z_1, z_2$  represent  $(z_2z_1z_0)_{2,3} = (x_1x_0)_{2,2} + (y_1y_0)_{2,2}$  (may require up to width 3)

*First approach:* half-adder for each column, then combine carry from right column with sum of left column

Write expressions for the circuit output values in terms of input values:

$z_0 =$  \_\_\_\_\_

$z_1 =$  \_\_\_\_\_

$z_2 =$  \_\_\_\_\_



*Second approach:* for middle column, first add carry from right column to  $x_1$ , then add result to  $y_1$

Write expressions for the circuit output values in terms of input values:

$z_0 =$  \_\_\_\_\_

$z_1 =$  \_\_\_\_\_

$z_2 =$  \_\_\_\_\_

*Extra example* Describe how to generalize this addition circuit for larger width inputs.



# Logical operators

Logical operators aka propositional connectives

<b>Conjunction</b>	AND	$\wedge$	<code>\land</code>	2 inputs	Evaluates to $T$ exactly when <b>both</b> inputs are $T$
<b>Exclusive or</b>	XOR	$\oplus$	<code>\oplus</code>	2 inputs	Evaluates to $T$ exactly when <b>exactly one</b> of inputs is $T$
<b>Disjunction</b>	OR	$\vee$	<code>\lor</code>	2 inputs	Evaluates to $T$ exactly when <b>at least one</b> of inputs is $T$
<b>Negation</b>	NOT	$\neg$	<code>\lnot</code>	1 input	Evaluates to $T$ exactly when its input is $F$

## Logical operators truth tables

Truth tables: Input-output tables where we use  $T$  for 1 and  $F$  for 0.

Input		Output		
		<b>Conjunction</b>	<b>Exclusive or</b>	<b>Disjunction</b>
$p$	$q$	$p \wedge q$	$p \oplus q$	$p \vee q$
$T$	$T$	$T$	$F$	$T$
$T$	$F$	$F$	$T$	$T$
$F$	$T$	$F$	$T$	$T$
$F$	$F$	$F$	$F$	$F$
				

Input	Output
$p$	<b>Negation</b> $\neg p$
$T$	$F$
$F$	$T$
	

## Logical operators example truth table

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

# Truth table to compound proposition

Given a truth table, how do we find an expression using the input variables and logical operators that has the output values specified in this table?

*Application:* design a circuit given a desired input-output relationship.

Input		Output	
$p$	$q$	$mystery_1$	$mystery_2$
$T$	$T$	$T$	$F$
$T$	$F$	$T$	$F$
$F$	$T$	$F$	$F$
$F$	$F$	$T$	$T$

Expressions that have output  $mystery_1$  are

Expressions that have output  $mystery_2$  are

## Dnf cnf definition

**Definition** An expression built of variables and logical operators is in **disjunctive normal form** (DNF) means that it is an OR of ANDs of variables and their negations.

**Definition** An expression built of variables and logical operators is in **conjunctive normal form** (CNF) means that it is an AND of ORs of variables and their negations.

## Dnf cnf example

*Extra example:* An expression that has output ? is:

Input			Output
$p$	$q$	$r$	?
$T$	$T$	$T$	$T$
$T$	$T$	$F$	$T$
$T$	$F$	$T$	$F$
$T$	$F$	$F$	$T$
$F$	$T$	$T$	$F$
$F$	$T$	$F$	$F$
$F$	$F$	$T$	$T$
$F$	$F$	$F$	$F$

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

## Logical operators full truth table

Input		Output				
$p$	$q$	Conjunction $p \wedge q$	Exclusive or $p \oplus q$	Disjunction $p \vee q$	Conditional $p \rightarrow q$	Biconditional $p \leftrightarrow q$
$T$	$T$	$T$	$F$	$T$	$T$	$T$
$T$	$F$	$F$	$T$	$T$	$F$	$F$
$F$	$T$	$F$	$T$	$T$	$T$	$F$
$F$	$F$	$F$	$F$	$F$	$T$	$T$
		“ $p$ 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  $p \rightarrow q$  false is to \_\_\_\_\_

The **hypothesis** of  $p \rightarrow q$  is \_\_\_\_\_ The **antecedent** of  $p \rightarrow q$  is \_\_\_\_\_

The **conclusion** of  $p \rightarrow q$  is \_\_\_\_\_ The **consequent** of  $p \rightarrow q$  is \_\_\_\_\_

## Converse inverse contrapositive

The **converse** of  $p \rightarrow q$  is \_\_\_\_\_

The **inverse** of  $p \rightarrow q$  is \_\_\_\_\_

The **contrapositive** of  $p \rightarrow q$  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 $(x \wedge y)$ , $(x \oplus y)$ , $(x \vee y)$ , $(x \rightarrow y)$ , $(x \leftrightarrow y)$

## 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

$$p \wedge F \equiv F$$

$$p \vee T \equiv T$$

$$p \wedge T \equiv p$$

$$p \vee F \equiv p$$

**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**

$$p \rightarrow q \equiv \neg p \vee q$$

$$p \rightarrow q \equiv \neg q \rightarrow \neg p$$

**Contrapositive**

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

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

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

*Extra examples:*

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

$p \rightarrow q$  is not logically equivalent to  $q \rightarrow p$  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**  $p \rightarrow q$  can be said in English as

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

**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.

## Consistency example

**Consistency:**

Whenever the system software is being upgraded, users cannot access the file system. If users can access the file system, then they can save new files. If users cannot save new files, then the system software is not being upgraded.

1. Translate to symbolic compound propositions
2. Look for some truth assignment to the propositional variables for which all the compound propositions output  $T$



# Algorithm redundancy

Real-life representations are often prone to corruption. Biological codes, like RNA, may mutate naturally<sup>1</sup> and during measurement; cosmic radiation and other ambient noise can flip bits in computer storage<sup>2</sup>. 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

---

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

---

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

---

```
1 procedure decode3( $c_{3k-1} \cdots c_0$ : 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  $a_{k-1} \cdots a_0$ 
```

---

Give a recursive definition of the set of outputs of the *redun3* procedure, *Out*,

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

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

---

<sup>1</sup>Mutations of specific RNA codons have been linked to many disorders and cancers.

<sup>2</sup>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  $A_1, A_2, \dots, A_n$ , 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,

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

## Algorithm rna mutation insertion deletion

Recall that  $S$  is defined as the set of all RNA strands, nonempty strings made of the bases in  $B = \{A, U, G, C\}$ . We define the functions

$$\begin{aligned} \text{mutation} : S \times \mathbb{Z}^+ \times B &\rightarrow S & \text{insertion} : S \times \mathbb{Z}^+ \times B &\rightarrow S \\ \text{deletion} : \{s \in S \mid \text{rlen}(s) > 1\} \times \mathbb{Z}^+ &\rightarrow S & & \text{with rules} \end{aligned}$$

---

```

1 procedure mutation( $b_1 \dots b_n$ : a RNA strand,  $k$ : a positive integer,  $b$ : an element of  $B$ )
2 for  $i := 1$  to  $n$ 
3   if  $i = k$ 
4      $c_i := b$ 
5   else
6      $c_i := b_i$ 
7 return  $c_1 \dots c_n$  {The return value is a RNA strand made of the  $c_i$  values}

```

---



---

```

1 procedure insertion( $b_1 \dots b_n$ : a RNA strand,  $k$ : a positive integer,  $b$ : an element of  $B$ )
2 if  $k > n$ 
3   for  $i := 1$  to  $n$ 
4      $c_i := b_i$ 
5    $c_{n+1} := b$ 
6 else
7   for  $i := 1$  to  $k-1$ 
8      $c_i := b_i$ 
9    $c_k := b$ 
10  for  $i := k+1$  to  $n+1$ 
11     $c_i := b_{i-1}$ 
12 return  $c_1 \dots c_{n+1}$  {The return value is a RNA strand made of the  $c_i$  values}

```

---



---

```

1 procedure deletion( $b_1 \dots b_n$ : a RNA strand with  $n > 1$ ,  $k$ : a positive integer)
2 if  $k > n$ 
3    $m := n$ 
4   for  $i := 1$  to  $n$ 
5      $c_i := b_i$ 
6 else
7    $m := n-1$ 
8   for  $i := 1$  to  $k-1$ 
9      $c_i := b_i$ 
10  for  $i := k$  to  $n-1$ 
11     $c_i := b_{i+1}$ 
12 return  $c_1 \dots c_m$  {The return value is a RNA strand made of the  $c_i$  values}

```

---

# 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 input so that  $\text{deletion}(\_, \_) = \text{G}$

## 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 $x$	Output		
	$V(x)$ $[x]_{2c,3} > 0$	$N(x)$ $[x]_{2c,3} < 0$	$Mystery(x)$
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 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?

# Predicate rna example

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

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

Truth set of  $H$  is \_\_\_\_\_

$F_A : S \rightarrow \{T, F\}$  defined recursively by:

Basis step:  $F_A(A) = T, F_A(C) = F_A(G) = F_A(U) = F$

Recursive step: If  $s \in S$  and  $b \in B$ , then  $F_A(sb) = F_A(s)$ .

Example where  $F_A$  evaluates to  $T$  is \_\_\_\_\_

Example where  $F_A$  evaluates to  $F$  is \_\_\_\_\_

## Rna rna len basecount definitions

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

Basis Step:  $A \in S, C \in S, U \in S, G \in S$

Recursive Step: If  $s \in S$  and  $b \in B$ , then  $sb \in S$

where  $sb$  is string concatenation.

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

		$rna len : S \rightarrow \mathbb{Z}^+$
Basis Step:	If $b \in B$ then	$rna len(b) = 1$
Recursive Step:	If $s \in S$ and $b \in B$ , then	$rna len(sb) = 1 + rna len(s)$

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

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

# Predicates example *rnalen* basecount

Using functions to define predicates:

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

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

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

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

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

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

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?

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

In English:

Witness that proves this existential quantification is true:

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

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  $T$ :

## Predicate notation

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



# 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

## 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**?