Discrete Mathematics Fundamentals

A Complete 10-Chapter Maths for Programmers Course

Topics covered:

Propositional & predicate logic · sets & set operations

Relations, equivalence relations & partial orders · functions & injectivity

Proof techniques & mathematical induction · combinatorics & the pigeonhole principle

Capstone: a full worked access-control system spanning every chapter

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Discrete Math Matters for Programmers

Discrete Mathematics Fundamentals

Chapter 1 · Why Discrete Math Matters for Programmers

A computer is, at every level, a discrete machine — a bit is either 0 or 1, never something in between; a list holds a specific, countable number of elements; a loop runs a specific, countable number of times; a program executes one distinct instruction, then the next. Discrete mathematics is the branch of math built specifically to reason about exactly this kind of world — countable, distinct, all-or-nothing — which is exactly why it sits underneath so much of computer science, whether or not the connection is ever made explicit.

Discrete vs. Continuous — What the Word Actually Means

"Discrete" and "continuous" describe two fundamentally different kinds of quantity, and most of mathematics falls cleanly into one camp or the other.

	DISCRETE	CONTINUOUS
Values	Distinct, separate, countable — you can list them one by one	Smoothly varying — between any two values, infinitely many more exist
Example quantity	The number of items in a shopping cart	A person's exact height
Core math tools	Logic, sets, combinatorics, graph theory	Calculus, real analysis
Where it shows up in code	Loop counts, array indices, boolean conditions, database rows	Physics simulations, gradient descent in machine learning

Both matter for programming — a machine learning course will eventually need calculus and linear algebra (both on this subject's own future list). This particular course is about the discrete side specifically: the math of things that are counted, not measured.

Five Concrete Connections to Code

Every topic in this course maps onto something you've almost certainly already used, whether or not it was ever named this way:

DISCRETE MATH TOPIC	WHERE IT ACTUALLY SHOWS UP
Propositional & predicate logic (Ch.2–3)	Every `if` statement, every boolean expression, every `WHERE` clause in SQL — all of it is propositional logic wearing different syntax
Sets (Ch.4)	Python's `set`, JavaScript's `Set`, SQL's `DISTINCT` and `UNION` — all directly implementing set operations
Relations & functions (Ch.5–6)	A database table *is* a relation in the mathematical sense; a hash map *is* a function from keys to values
Proof technique & induction (Ch.7–8)	Arguing a recursive function terminates correctly, or that a loop invariant genuinely holds on every pass, is a proof by induction whether or not anyone calls it one
Combinatorics (Ch.9)	"How many possible states could this system be in" underlies complexity analysis, test-case coverage, and cryptographic key-space size

What This Course Won't Cover

Several genuinely related topics are deliberately left for their own future courses under this same Maths for Programmers subject, rather than folded in here as extra chapters:

- **Graph Theory** — networks, trees, and pathfinding get their own dedicated course, even though a graph is technically also a kind of relation (Chapter 5's own territory)
- **Boolean Algebra & Digital Logic** — logic gates and circuit-level reasoning get their own dedicated course, even though propositional logic (Chapter 2) is the direct mathematical foundation underneath them
- **Algorithms & Complexity** — Big-O analysis gets its own dedicated course, even though combinatorics (Chapter 9) is exactly the counting math that analysis depends on

> **WHY DRAW THE LINE HERE INSTEAD OF COVERING EVERYTHING AT ONCE**
>
> Each of those three topics is substantial enough to deserve its own real depth, not a single rushed chapter bolted onto this course. This course stays tightly scoped to logic, sets, relations, and combinatorics — the direct foundation the other three will each build on, once their own turn comes.

Where This Course Is Headed

CHAPTER	TOPIC
2	Propositional Logic — Statements, Connectives & Truth Tables
3	Predicate Logic & Quantifiers
4	Sets & Set Operations

CHAPTER	TOPIC
5	Relations — Properties, Equivalence Relations & Partial Orders
6	Functions — Injective, Surjective & Bijective
7	Proof Techniques — Direct, Contrapositive & Contradiction
8	Mathematical Induction
9	Combinatorics — Counting, Permutations & Combinations
10	Capstone — Applying Discrete Math to Real Programming Problems

THIS COURSE'S THROUGHLINE

Every chapter answers a version of the same question: what's the precise, formal way to say something you've probably already been doing by instinct? Once the formal vocabulary is in place, it turns out to make reasoning about code — correctness, edge cases, "how many ways could this go wrong" — noticeably more rigorous than intuition alone.

Hands-On Exercises

EXERCISE 1

Classify each of the following as discrete or continuous, and briefly justify each answer: (a) the number of users currently logged into a website, (b) the exact CPU temperature at a given instant, (c) the number of rows returned by a SQL query, (d) the amount of time a function took to execute.

View solution

EXERCISE 2

A colleague claims "math for programmers" should just mean calculus and linear algebra, since that's what machine learning courses always cover first. Using this chapter's own five connections, explain what specifically would be missing from a programmer's toolkit if discrete math were skipped entirely.

View solution

EXERCISE 3

For each of the following real code artifacts, name which discrete math topic from this chapter's own table it most directly maps to, and explain the connection in one or two sentences: (a) a Python `set` used to remove duplicate values from a list, (b) a SQL table with a `UNIQUE` constraint on an email column, (c) an `if` statement with three chained `and` / `or` conditions.

📄 View solution

CHAPTER 1 QUICK REFERENCE

- **Discrete** = countable, distinct values; **continuous** = smoothly varying values — computers are fundamentally discrete machines
- Five direct connections: logic → conditionals, sets → collection types, relations → databases/hash maps, proof/induction → correctness arguments, combinatorics → complexity/keyspace counting
- Deliberately out of scope here: Graph Theory, Boolean Algebra & Digital Logic, and Algorithms & Complexity each get their own future course
- This course stays tightly scoped to logic, sets, relations, and combinatorics — the direct foundation those three future courses will each build on
- **Next chapter:** Propositional Logic — statements, connectives, and truth tables

CHAPTER 2 OF 10

Propositional Logic: Statements, Connectives & Truth Tables

Discrete Mathematics Fundamentals

Chapter 2 · Propositional Logic: Statements, Connectives & Truth Tables

Chapter 1 claimed that "every `if` statement, every boolean expression ... is propositional logic wearing different syntax." This chapter makes that literal — the formal system for reasoning about statements that are simply true or false, and the exact operations that combine them: the same operations as `&&`, `||`, and `!`, just written with mathematical symbols instead of code syntax.

What Counts as a Proposition

A **proposition** is a statement that is unambiguously either true or false — never both, never neither, and never a matter of opinion.

STATEMENT	IS IT A PROPOSITION?
`5 > 3`	Yes — true
"Paris is the capital of France"	Yes — true
"What time is it?"	No — a question, not a statement that's true or false
`x > 3`	No — not yet. Its truth depends on a value `x` hasn't been given. This is a **predicate**, and Chapter 3 is entirely about it.

The Five Core Connectives

Propositions combine using **connectives** — and every one of them already has a direct equivalent in code you've used before.

CONNECTIVE	SYMBOL	MEANING	CODE EQUIVALENT
NOT	¬p	True exactly when p is false	`not p` / `!p`
AND	p ∧ q	True only when both are true	`p and q` / `p && q`

CONNECTIVE	SYMBOL	MEANING	CODE EQUIVALENT
OR	p ∨ q	True when at least one is true	`p or q` / `p || q`
XOR	p ⊕ q	True when exactly one is true, not both	`p != q` (on booleans) / `p ^ q`
IMPLIES	p → q	False only when p is true and q is false	No direct operator — built from `(not p) or q`

Truth Tables — Defining a Connective by Every Possible Case

A truth table lists every combination of inputs and the resulting output — the formal, complete definition of what a connective actually means.

P	¬P
T	F
F	T

P	Q	P ∧ Q	P ∨ Q	P ⊕ Q	P → Q
T	T	T	T	F	T
T	F	F	T	T	F
F	T	F	T	T	T
F	F	F	F	F	T

THE COUNTERINTUITIVE ROW: IMPLIES IS TRUE WHEN P IS FALSE

"If p, then q" is considered **true** whenever p is false — regardless of q. "If pigs can fly, then 2 + 2 = 5" is, by this definition, a true statement, since the premise is false. This isn't a trick — it's why a guard clause like `if not condition: return` works the way it does: when the precondition doesn't hold, the rest of the logic is vacuously satisfied, because there was never a case to check in the first place.

De Morgan's Laws — Simplifying Negated Conditions

Two of the most practically useful identities in this entire chapter, both directly checkable against the truth tables above:

LAW	STATEMENT
De Morgan's (AND)	¬(p ∧ q) ≡ ¬p ∨ ¬q

LAW	STATEMENT
De Morgan's (OR)	¬(p ∨ q) ≡ ¬p ∧ ¬q

In plain terms: negating an AND flips it to an OR of the negations, and negating an OR flips it to an AND of the negations — the connective itself flips, not just the individual pieces.

```
# These two conditions are logically identical — De Morgan's (AND):
if not (a and b):
    ...

if (not a) or (not b):
    ...

# A genuinely common mistake — this is NOT the same thing:
if not a and not b:   # this is ¬a ∧ ¬b, a completely different condition
```

WHY THIS MATTERS IN REAL CODE

`not a and not b` looks like a natural way to write "not both" but it's actually the truth table for "neither" — a genuinely different condition. De Morgan's Law is exactly the tool that tells you the correct rewrite of `not (a and b)` is `(not a) or (not b)`, not `(not a) and (not b)`.

Logical Equivalence

Two propositions are **logically equivalent** (written ≡) when they produce the same output for every possible combination of inputs — verified by comparing their truth tables row by row. De Morgan's Laws above are exactly this: two differently-written expressions that are provably, always, the same thing. This matters directly for refactoring — if two boolean expressions are logically equivalent, replacing one with the other can never change what your program does.

A QUICK WAY TO DOUBLE-CHECK P → Q ≡ ¬P ∨ Q

Compare the IMPLIES column above (T, F, T, T) against a truth table for ¬p ∨ q: row by row, ¬p is F, F, T, T and q is T, F, T, F, so ¬p ∨ q comes out T, F, T, T — an exact match, row for row. This is exactly how you'd verify any claimed equivalence: build both truth tables, compare every row.

Short-Circuit Evaluation — A Runtime Detail, Not a Logic Change

Python and JavaScript's `and` / `or` (`&&` / `||`) don't always evaluate both sides — if the left side of `and` is already false, the right side is never even run, since the overall result is already determined. This is called **short-circuit evaluation**. It doesn't change the truth table at all — the logical result is identical either way —

but it does matter if the right-hand side has a side effect (like calling a function), since that side effect might never actually happen.

Hands-On Exercises

EXERCISE 1

Build a complete truth table for the compound proposition `(p ∨ q) ∧ ¬p`, and state in plain English what condition it actually captures.

View solution

EXERCISE 2

A function contains the condition `if not (is_valid and is_authorized):`. Using De Morgan's Law, rewrite this without a negated compound expression, and explain why your rewrite is guaranteed to behave identically for every possible combination of `is_valid` and `is_authorized`.

View solution

EXERCISE 3

Determine whether `p → q` and `q → p` are logically equivalent, by building both truth tables and comparing them row by row. If they're not equivalent, give a concrete real-world example of a true "if p then q" statement whose reverse is false.

View solution

CHAPTER 2 QUICK REFERENCE

- A **proposition** is a statement that's unambiguously true or false — a statement with an unknown variable is a *predicate* instead (Chapter 3)
- Five connectives: NOT (¬), AND (∧), OR (∨), XOR (⊕), IMPLIES (→) — each with a direct code equivalent
- A truth table defines a connective completely, by listing the output for every combination of inputs
- **p → q is true whenever p is false**, regardless of q — the source of "vacuous truth"
- **De Morgan's Laws:** ¬(p ∧ q) ≡ ¬p ∨ ¬q, and ¬(p ∨ q) ≡ ¬p ∧ ¬q — the connective itself flips
- Logically equivalent expressions can always be swapped for each other without changing program behavior
- Short-circuit evaluation is a runtime detail — it doesn't change the logical truth table, only whether side effects on the right-hand side actually run

- **Next chapter:** Predicate logic and quantifiers — what happens once a statement has a variable in it

CHAPTER 3 OF 10

Predicate Logic & Quantifiers

Discrete Mathematics Fundamentals

Chapter 3 · Predicate Logic & Quantifiers

Chapter 2 flagged `x > 3` as not a proposition — its truth depends on a variable that hasn't been given a value yet. This chapter covers exactly that case properly: **predicates**, and the two **quantifiers** that turn a predicate into an actual, honest-to-goodness proposition.

What a Predicate Actually Is

A predicate is a statement containing one or more variables, which becomes a genuine proposition — true or false — the moment those variables are given specific values. Written `P(x)`, a predicate is a template for a proposition, not a proposition itself.

EXPRESSION	WHAT IT IS
`P(x): x > 3`	A predicate — no fixed truth value yet
`P(5)`	A proposition — true
`P(2)`	A proposition — false

This is already exactly what a boolean-returning function is in code — a predicate, parameterized truth, written in a different notation:

```
def is_adult(age):
    return age >= 18

# is_adult is the predicate P(x). is_adult(20) is a proposition (True).
```

The Universal Quantifier — ∀ ("for all")

`∀x P(x)` means P(x) is true for *every* value of x in whatever domain you're considering. This is exactly what Python's `all()` computes.

∀x P(x) — "for every x, P(x) holds" all(is_adult(age) for age in ages) # True only if every single age satisfies is_adult

The Existential Quantifier — ∃ ("there exists")

`∃x P(x)` means P(x) is true for *at least one* value of x in the domain — not necessarily all, just one is enough. This is exactly what Python's `any()` computes.

∃x P(x) — "there exists an x such that P(x) holds" any(is_adult(age) for age in ages) # True if at least one age satisfies is_adult

A QUANTIFIER IS MEANINGLESS WITHOUT A STATED DOMAIN

`∀x P(x)` only makes sense once you've said what x ranges over — "for all x" in what set, exactly? In code, the domain is simply whatever you're iterating over — `ages` in the examples above. Leaving the domain unstated (in math or in code) is a genuine, common source of confusion.

Combining Quantifiers: "Every P Is a Q" vs. "Some P Is a Q"

In practice, quantifiers almost always pair with a specific second connective, and the pairing is easy to get backwards:

ENGLISH	CORRECT TRANSLATION	WHY THIS CONNECTIVE
"Every P is a Q"	∀x (P(x) → Q(x))	For any x, *if* it's a P, *then* it must also be a Q
"Some P is a Q"	∃x (P(x) ∧ Q(x))	There's an x that is *both* a P *and* a Q, at the same time

THE CLASSIC TRANSLATION MISTAKE

Writing `∀x (P(x) ∧ Q(x))` for "every P is a Q" is a genuine, common error — it actually claims that *every single x in the entire domain* is both a P and a Q, which is a far stronger (and usually false) statement. "Every student who submitted on time gets full marks" does **not** mean every person in existence submitted on time *and* got full marks — it means *if* someone submitted on time, *then* they got full marks. ∀ pairs with →; ∃ pairs with ∧ — mixing these up is the single most common predicate-logic translation mistake.

Negating Quantified Statements

Predicate logic has its own version of Chapter 2's De Morgan's Laws — negating a quantifier flips it to the other quantifier:

STATEMENT	NEGATION	PLAIN ENGLISH
¬(∀x P(x))	≡ ∃x ¬P(x)	"Not all" means "at least one fails"
¬(∃x P(x))	≡ ∀x ¬P(x)	"None exist" means "all fail"

```
# These two are logically equivalent — a genuinely useful real refactor: not all(is_verified(u) for u in users)
any(not is_verified(u) for u in users)
```

COMBINING THIS WITH CHAPTER 2'S OWN ¬(P → Q) RULE

Negating "every P is a Q" — ¬(∀x (P(x) → Q(x))) — becomes ∃x ¬(P(x) → Q(x)), and Chapter 2 already established that ¬(p → q) ≡ p ∧ ¬q. Chaining both rules together: **¬(∀x (P(x) → Q(x))) ≡ ∃x (P(x) ∧ ¬Q(x))** — "not every P is a Q" means "there's a P that is *not* a Q." This is exactly the two-step reasoning a proof by contradiction (Chapter 7) often starts with: to disprove "every P is a Q," it's enough to produce a single counterexample — one P that isn't a Q.

Nested Quantifiers — Order Genuinely Changes the Meaning

When two quantifiers appear together, swapping their order can produce a completely different statement:

STATEMENT	MEANING
∀x ∃y Mother(y, x)	Every person x has *some* mother y — a possibly different y for each x
∃y ∀x Mother(y, x)	There's *one single* y who is the mother of every person x

The first is an ordinary true fact about the world. The second claims one person is literally everyone's mother — obviously false. Same symbols, same predicate, reversed order, opposite truth value. When both quantifiers are the same type (∀∀ or ∃∃), order doesn't matter; the moment they're mixed (∀∃ or ∃∀), it does.

Hands-On Exercises

EXERCISE 1

Translate "every function in this module has a docstring" into quantified predicate logic, clearly stating the domain and which two predicates you're using. Then explain why using ∧ instead of → would be a translation mistake here.

View solution

EXERCISE 2

A test asserts `not any(order.total < 0 for order in orders)`. Rewrite this using De Morgan's Law for quantifiers so it reads as an ∀ statement instead of a negated ∃ statement, and state in plain English what the rewritten version says.

View solution

EXERCISE 3

For the two statements ∀x ∃y Likes(x, y) ("everyone likes someone") and ∃y ∀x Likes(x, y) ("there's someone everyone likes"), explain in your own words why these are genuinely different claims, and give an example of a group of people for which the first is true but the second is false.

View solution

CHAPTER 3 QUICK REFERENCE

- A **predicate** `P(x)` becomes a proposition once `x` is given a value — exactly a boolean-returning function in code
- **∀x P(x)** ("for all") ≡ `all(P(x) for x in domain)`
- **∃x P(x)** ("there exists") ≡ `any(P(x) for x in domain)`
- A quantifier means nothing without a stated domain — in code, that's whatever you're iterating over
- **"Every P is Q"** → ∀x (P(x) → Q(x)); **"Some P is Q"** → ∃x (P(x) ∧ Q(x)) — ∀ pairs with →, ∃ pairs with ∧
- **Negation flips the quantifier:** ¬∀x P(x) ≡ ∃x ¬P(x); ¬∃x P(x) ≡ ∀x ¬P(x)
- Nested quantifier order matters when the two quantifiers differ (∀∃ vs. ∃∀) — same symbols, opposite meaning
- **Next chapter:** Sets and set operations

CHAPTER 4 OF 10

Sets & Set Operations

Discrete Mathematics Fundamentals

Chapter 4 · Sets & Set Operations

Chapter 1 named sets as directly underlying "Python's `set`, JavaScript's `Set`, SQL's `DISTINCT` and `UNION`." This chapter covers set theory properly — what a set actually is, the notation mathematicians use for it, the core operations, and how directly each one maps onto code you've almost certainly already written.

What a Set Actually Is

A set is an **unordered collection of distinct elements** — no duplicates, and no built-in notion of order. `{1, 2, 3}` and `{3, 1, 2}` are the exact same set. Membership is written `x ∈ A` ("x is an element of A") or `x ∉ A` ("x is not an element of A"). The **empty set**, written `∅` or `{}`, contains no elements at all.

Set-Builder Notation — A Predicate in Disguise

`{x | P(x)}` reads as "the set of all x such that P(x) holds" — and P(x) is exactly Chapter 3's own predicate. A set-builder expression is literally a predicate, repurposed to define which elements belong to a set rather than to just answer true/false.

{x | x ∈ range(10) ∧ x is even} — set-builder notation {x for x in range(10) if x % 2 == 0} # {0, 2, 4, 6, 8} — Python's set comprehension is set-builder notation, verbatim

Core Set Operations

OPERATION	SYMBOL	MEANING	PYTHON EQUIVALENT
Union	A ∪ B	Everything in A, B, or both	`A | B`
Intersection	A ∩ B	Only what's in both A and B	`A & B`
Difference	A – B	In A, but not in B	`A - B`
Symmetric difference	A Δ B	In exactly one of A or B, not both	`A ^ B`

Worked Example: Two Sets of Numbers

Let `A = {1, 2, 3, 4, 5}` and `B = {4, 5, 6, 7}`:

EXPRESSION	RESULT
A ∪ B	{1, 2, 3, 4, 5, 6, 7}
A ∩ B	{4, 5}
A – B	{1, 2, 3}
B – A	{6, 7}
A Δ B	{1, 2, 3, 6, 7}

Note that `A - B` and `B - A` are different sets — difference isn't symmetric, unlike union and intersection.

Subsets — Defined Directly by Chapter 3's Own Quantifier

`A ⊆ B` ("A is a subset of B") means every element of A is also in B. This has a precise quantified definition, using exactly the pattern from Chapter 3:

A ⊆ B ≡ ∀X (X ∈ A → X ∈ B)

This is Chapter 3's own "every P is a Q" pattern, applied directly: for any x at all, *if* x is in A, *then* x must also be in B. `A ⊂ B` (proper subset) adds one more condition: A ⊆ B *and* A ≠ B — every element of A is in B, but B has at least one element A doesn't.

Cardinality & the Power Set

Cardinality, written `|A|`, is simply the number of elements in a finite set — exactly what `len()` computes in Python. The **power set**, written `P(A)`, is the set of *every possible subset* of A, including the empty set and A itself.

A = {1, 2} — every possible subset: ∅, {1}, {2}, {1, 2} # 4 subsets total # |P(A)| = 2^|A| → 2^2 = 4 — checks out

This `2^|A|` result isn't a coincidence — it's a direct combinatorics fact (Chapter 9's own territory): each element independently either is or isn't in a given subset, a binary choice repeated once per element.

Sets Are Unordered *and* Deduplicated — Both at Once

CONVERTING A LIST TO A SET LOSES TWO THINGS, NOT ONE

`list(set([3, 1, 2, 2, 1]))` both removes duplicates (a set can't contain the same element twice by definition) and discards the original order (a set has no notion of order at all — it isn't merely "unsorted," order genuinely isn't part of what a set *is*). Python's own documentation is explicit that sets are unordered; the specific order you happen to see when iterating one is an implementation detail, not a guarantee, and can even change between separate runs of the same program due to hash randomization. Never rely on set iteration order for anything that needs to be reproducible.

Hands-On Exercises

EXERCISE 1

Given `A = {2, 4, 6, 8, 10}` and `B = {4, 8, 12, 16}`, compute A ∪ B, A ∩ B, A – B, and B – A.

View solution

EXERCISE 2

For the set `A = {a, b, c}`, list every element of the power set P(A) explicitly, and confirm the total matches `2^|A|`. Then write out the quantified definition of `A ⊆ B` from memory and explain, in your own words, why it uses → rather than ∧.

View solution

EXERCISE 3

A marketing team has set E (users who opted into email) and set P (users who made a purchase in the last 30 days). For each request below, name the correct set operation: (a) "everyone who purchased but hasn't opted into email," (b) "everyone who either purchased, opted in, or both," (c) "everyone who both purchased and opted in."

View solution

CHAPTER 4 QUICK REFERENCE

- A set is an **unordered** collection of **distinct** elements — no duplicates, no built-in order
- Set-builder notation `{x | P(x)}` is a predicate defining membership — directly what a Python set comprehension is
- ∪ union (`|`), ∩ intersection (`&`), **–** difference (`-`), **Δ** symmetric difference (`^`)
- **$A \subseteq B \equiv \forall x\,(x \in A \rightarrow x \in B)$** — every element of A is also in B, using Chapter 3's own "every P is Q" pattern
- **|A|** = cardinality = `len()`; **P(A)** = power set = every subset, with `|P(A)| = 2^|A|`

- Converting a list to a set discards both duplicates and order — never rely on set iteration order for anything reproducible
- **Next chapter:** Relations — properties, equivalence relations, and partial orders

CHAPTER 5 OF 10

Relations: Properties, Equivalence Relations & Partial Orders

Discrete Mathematics Fundamentals

Chapter 5 · Relations: Properties, Equivalence Relations & Partial Orders

Chapter 1 claimed "a database table is a relation in the mathematical sense." This chapter makes that precise — what a relation actually is formally, the four properties that classify how a relation behaves, and two especially important categories: equivalence relations (which underlie grouping and equality) and partial orders (which underlie dependency resolution).

What a Relation Actually Is

Formally, a relation `R` from set A to set B is a **subset of the Cartesian product A × B** — the set of every possible ordered pair `(a, b)` with `a ∈ A` and `b ∈ B`. A relation "on" a single set A is a subset of A × A. Written `(a, b) ∈ R`, or more commonly `a R b` — the same way `a < b` or `a ≡ b` are written.

A database table *is* exactly this — a table with columns `(user_id, order_id)` is a subset of all possible `(user_id, order_id)` pairs, selecting out exactly the ones that actually hold. A "friends" table in a social network is a relation on the set of users.

Four Key Properties

Each property is a quantified statement, following Chapter 3's own notation directly:

PROPERTY	DEFINITION	HOLDS FOR	FAILS FOR
Reflexive	∀a (a R a)	≤ — a ≤ a is always true	< — a < a is never true
Symmetric	∀a ∀b (a R b → b R a)	"is a sibling of"	"is a parent of"
Transitive	∀a ∀b ∀c ((a R b ∧ b R c) → a R c)	<, ≤, =	"is a friend of" — a friend of a friend isn't necessarily a friend
Antisymmetric	∀a ∀b ((a R b ∧ b R a) → a = b)	≤ — a≤b and b≤a forces a=b	"is a friend of" — mutual friendship doesn't force the two people to be the same person

"NOT SYMMETRIC" AND "ANTISYMMETRIC" ARE NOT OPPOSITES

A relation can genuinely be neither symmetric nor antisymmetric, or — as this chapter's own first exercise shows — antisymmetric without being symmetric at all. Antisymmetric doesn't mean "never symmetric anywhere" — it only forbids a R b and b R a both holding *unless* a and b are the same element. The diagonal (a R a) never violates it.

Equivalence Relations — Reflexive + Symmetric + Transitive

A relation with all three of the first properties together is an **equivalence relation** — a formal notion of "these things count as the same" for some specific purpose. An equivalence relation splits its entire set into disjoint **equivalence classes**, where everything inside one class is considered equivalent to everything else in that same class.

THIS IS EXACTLY WHAT GROUP BY DOES

"Has the same email domain as" is a genuine equivalence relation on a set of users — reflexive (everyone shares a domain with themselves), symmetric (if A shares a domain with B, B shares it with A), and transitive (if A and B share a domain, and B and C share a domain, A and C do too). The equivalence classes are exactly the groups a SQL `GROUP BY email_domain` would produce — each class is one specific domain's worth of users. Grouping data by a shared value *is* partitioning by an equivalence relation, whether or not the query was ever thought of that way.

Partial Orders — Reflexive + Antisymmetric + Transitive

Swap symmetric for antisymmetric, and a different, equally important category appears: a **partial order**. It describes an ordering where, unlike numbers under ≤, not every pair of elements necessarily needs to be comparable at all.

"Package A depends on package B" (or its reverse, "must be installed before") is a classic partial order — reflexive in a trivial sense, antisymmetric (if A must come before B and B must come before A, something's actually wrong — normally impossible in a valid dependency graph), and transitive (a chain of dependencies carries through). Crucially, two unrelated packages might have no dependency relationship between them at all — neither has to come before the other, which is exactly what makes it *partial* rather than a total order like ≤ on numbers, where any two numbers are always comparable.

Subset (`⊆`, Chapter 4) is another classic partial order — two arbitrary sets aren't always comparable by ⊆ either.

Hands-On Exercises

EXERCISE 1

On the set {1, 2, 3}, consider the relation R = {(1,1), (2,2), (3,3), (1,2)}. Determine whether R is reflexive, symmetric, transitive, and antisymmetric, justifying each answer.

View solution

EXERCISE 2

Prove that "has the same last two digits of their employee ID" is an equivalence relation on a set of employees, by explicitly checking all three required properties. Then describe what the resulting equivalence classes would actually look like.

View solution

EXERCISE 3

The relation "a divides b" (written a | b, meaning b is a whole-number multiple of a) is defined on the positive integers. Determine whether this is a partial order by checking reflexivity, antisymmetry, and transitivity, and give one example pair of positive integers that are *not* comparable under this relation (neither divides the other).

View solution

CHAPTER 5 QUICK REFERENCE

- A relation is a subset of a Cartesian product — a database table *is* a relation, literally
- **Reflexive:** ∀a(a R a) — **Symmetric:** ∀a∀b(a R b → b R a) — **Transitive:** ∀a∀b∀c((a R b ∧ b R c) → a R c) — **Antisymmetric:** ∀a∀b((a R b ∧ b R a) → a=b)
- **Equivalence relation** = reflexive + symmetric + transitive — partitions a set into equivalence classes, exactly what `GROUP BY` does
- **Partial order** = reflexive + antisymmetric + transitive — an ordering where not every pair needs to be comparable, unlike ≤ on numbers
- "Not symmetric" and "antisymmetric" are not opposites — a relation can be antisymmetric without ever being symmetric anywhere off the diagonal
- **Next chapter:** Functions — injective, surjective, and bijective

CHAPTER 6 OF 10

Functions: Injective, Surjective & Bijective

Discrete Mathematics Fundamentals

Chapter 6 · Functions: Injective, Surjective & Bijective

Chapter 5 covered relations broadly. This chapter covers a specific, exceptionally important kind of relation: a **function** — and Chapter 1 already named its code equivalent directly: "a hash map is a function from keys to values."

What Makes a Relation a Function

A relation `f` from A to B is a function if **every element of A maps to exactly one element of B** — not zero, not more than one, exactly one. A relation that sends some input to two different outputs simply isn't a function at all.

> **THIS IS EXACTLY WHY A DICT KEY CAN ONLY HOLD ONE VALUE**
>
> A Python dictionary *structurally enforces* the function property — it physically cannot represent a non-function relation. `d[key] = value1` followed by `d[key] = value2` doesn't create two mappings for the same key; it overwrites the first, because a dict is a direct implementation of "each key maps to exactly one value." You couldn't accidentally build a non-function with a dict even if you tried.

Domain, Codomain & Range

Three related but genuinely different sets, easy to mix up:

TERM	MEANING
Domain	The full set of allowed inputs
Codomain	The set outputs are *declared* to come from
Range (image)	The set of outputs that *actually occur* — always a subset of the codomain, sometimes a smaller one

Example: `f(x) = x²` declared as a function from all integers to all integers has codomain = all integers, but its range is only the non-negative perfect squares (0, 1, 4, 9, 16, ...) — the codomain is bigger than what's actually produced. This gap between codomain and range is exactly what the next section formalizes.

Injective (One-to-One)

A function is **injective** when different inputs always produce different outputs — no two distinct inputs ever collide on the same output. Formally: `∀a₁ ∀a₂ (f(a₁) = f(a₂) → a₁ = a₂)`.

Surjective (Onto)

A function is **surjective** when every element of the codomain actually gets hit by something — the range equals the entire codomain, with nothing left over. Formally: `∀b ∈ B ∃a ∈ A (f(a) = b)`.

Three Small, Fully Verifiable Examples

FUNCTION	INJECTIVE?	SURJECTIVE?
A={1,2}, B={a,b,c}: f(1)=a, f(2)=b	Yes — 1 and 2 map to different outputs	No — c is never hit
A={1,2,3}, B={a,b}: f(1)=a, f(2)=a, f(3)=b	No — f(1)=f(2) but 1≠2	Yes — both a and b are hit
A={1,2,3}, B={x,y,z}: f(1)=x, f(2)=y, f(3)=z	Yes	Yes

Bijective — Both at Once

A function that's both injective and surjective is **bijective** — a perfect one-to-one correspondence between every element of A and every element of B, nothing left unmatched on either side. The third example above is bijective. A bijective function is exactly what "invertible" means: because every output came from exactly one input, you can always walk backward from output to input unambiguously.

> **WHY THIS MATTERS FOR ENCODING SCHEMES**
>
> An encoding function where `decode(encode(x))` always recovers the original `x` exactly requires `encode` to be injective at minimum — if two different inputs ever produced the same encoded output, decoding that output could never reliably tell you which original input it came from.

Why a Cardinality Mismatch Rules Out Some Functions Entirely

For finite sets, a bijection between A and B can only exist when `|A| = |B|` exactly. If `|A| > |B|`, no injective function from A to B can exist at all — this is the **pigeonhole principle**, formalized properly in Chapter 9.

> **THIS IS EXACTLY WHY HASH COLLISIONS ARE MATHEMATICALLY UNAVOIDABLE**

A hash function maps an effectively unlimited number of possible input strings down to a fixed-size output — say, a 32-bit hash, which has exactly 2^{32} possible values. Since the set of possible inputs is vastly larger than 2^{32}, no hash function from strings to 32-bit values can ever be injective, no matter how cleverly it's designed. Collisions aren't a flaw in a specific hash function's implementation — they're a direct, unavoidable consequence of mapping a larger set into a smaller one.

Hands-On Exercises

EXERCISE 1

For A = {1, 2, 3, 4} and B = {p, q}, with f(1)=p, f(2)=q, f(3)=p, f(4)=q, determine whether f is injective, surjective, both, or neither, and justify each part of your answer.

View solution

EXERCISE 2

Consider two mappings on a set of users: (a) each user's unique employee ID maps to that user's record, and (b) each user's country of residence maps to that user's record. For each, state whether the mapping is injective, and explain the practical consequence for whether you could reliably look up "the" user just from the output alone.

View solution

EXERCISE 3

A system needs to assign a unique 4-digit PIN (10,000 possible values) to each of 15,000 new users. Using this chapter's own cardinality reasoning, explain why no injective assignment of PINs to users is possible here, regardless of how the PINs are chosen.

View solution

CHAPTER 6 QUICK REFERENCE

- A function maps every input to exactly one output — a dict enforces this structurally, it can't represent anything else
- **Domain** = allowed inputs, **codomain** = declared output set, **range** = outputs that actually occur (⊆ codomain)
- **Injective:** different inputs always give different outputs — **Surjective:** every codomain element gets hit
- **Bijective** = both at once = a perfect one-to-one correspondence = invertible
- For finite sets, a bijection requires |A| = |B| exactly; |A| > |B| rules out any injective function — this is why fixed-size hash functions can never avoid collisions

- **Next chapter:** Proof techniques — direct, contrapositive, and contradiction

CHAPTER 7 OF 10

Proof Techniques: Direct, Contrapositive & Contradiction

Discrete Mathematics Fundamentals

Chapter 7 · Proof Techniques: Direct, Contrapositive & Contradiction

Every chapter so far has justified its claims informally. This chapter makes the reasoning itself the subject — the standard techniques for proving a claim is actually, unconditionally true, not just true in every case anyone happened to check.

What a Proof Actually Is — and Why Testing Isn't One

A proof is a logically valid chain of steps from accepted facts (definitions, already-proven results) to the claim being made, with no gaps — every step follows necessarily from what came before.

PASSING EVERY TEST CASE IS EVIDENCE, NOT PROOF

Running a function against a thousand test cases and seeing it pass every time is real, useful evidence — but it's not the same claim as "this function is correct for all possible inputs." A single untested input could still break it. This is exactly why Chapter 8's mathematical induction matters: it's the one technique in this course that can genuinely prove a property holds for *every* case in an infinite family, not just the finitely many cases anyone got around to checking.

Direct Proof

To prove "if P then Q," assume P is true, and show through a valid chain of reasoning that Q must then also be true.

CLAIM: IF N IS EVEN, THEN N^2 IS EVEN

Proof. Assume n is even. By definition, $n = 2k$ for some integer k. Then $n^2 = (2k)^2 = 4k^2 = 2(2k^2)$. Since $2k^2$ is an integer, n^2 is 2 times an integer — which is exactly the definition of even. ■

Proof by Contrapositive

Chapter 2 established that `p → q ≡ ¬q → ¬p` — the contrapositive is *logically equivalent* to the original implication, so proving one genuinely proves the other. This matters because the contrapositive direction is

sometimes much easier to work with directly.

CLAIM: IF N^2 IS ODD, THEN N IS ODD

Proving this directly is awkward. Its contrapositive, though, is "if n is *not* odd (i.e., even), then n^2 is *not* odd (i.e., even)" — which is exactly the direct proof already given above. Since the contrapositive is proven, and it's logically equivalent to the original claim, the original claim is proven too, with no additional work. ■

Proof by Contradiction

Assume the claim is *false*, then show this assumption leads to something logically impossible. Since a false starting assumption can't lead to a genuine contradiction unless the assumption itself was the problem, the original claim must actually be true.

CLAIM: √2 IS IRRATIONAL (CANNOT BE WRITTEN AS A RATIO OF INTEGERS)

Proof. Assume, for contradiction, that √2 *is* rational. Then $\sqrt{2} = a/b$ for some integers a, b with no common factor (in lowest terms). Squaring both sides: $2 = a^2/b^2$, so $a^2 = 2b^2$ — meaning a^2 is even.

By this chapter's own contrapositive result above, if a^2 is even, then a is even. So $a = 2k$ for some integer k. Substituting: $(2k)^2 = 2b^2$, so $4k^2 = 2b^2$, so $b^2 = 2k^2$ — meaning b^2 is even too, and by the same result, b is even.

But if both a and b are even, they share a common factor of 2 — directly contradicting the assumption that a/b was already in lowest terms. This is a genuine contradiction. The only assumption that could be wrong is the starting one — so √2 is not rational. ■

THIS PROOF REUSES THE CHAPTER'S OWN EARLIER RESULT

The "a^2 even → a even" step above is exactly the contrapositive result proven earlier in this same chapter — proofs build on each other constantly, the same way functions call other functions rather than reimplementing everything from scratch each time.

Choosing a Technique

USE THIS WHEN...	TECHNIQUE
The forward reasoning from P to Q is already straightforward	Direct

USE THIS WHEN...	TECHNIQUE
Reasoning from ¬Q backward is cleaner than reasoning from P forward	**Contrapositive**
Proving "there is no X" or an impossibility claim directly seems hard	**Contradiction**

THE DEBUGGING PARALLEL

"Assume the bug isn't in this function, trace through what that would mean, and find it leads somewhere impossible" is genuinely the same underlying logic as proof by contradiction — assuming the opposite of what you suspect, and using the resulting absurdity as evidence for where the real problem actually is.

Hands-On Exercises

EXERCISE 1

Give a direct proof that if n is odd, then n^2 is odd.

View solution

EXERCISE 2

Prove "if n^2 is even, then n is even" using proof by contrapositive. (Hint: what does the contrapositive of this specific statement look like, and does Exercise 1's own result already prove it?)

View solution

EXERCISE 3

Use proof by contradiction to show that there is no smallest positive rational number.

View solution

CHAPTER 7 QUICK REFERENCE

- A proof is a gapless logical chain from accepted facts to a conclusion — testing many cases is evidence, not proof
- **Direct:** assume P, show Q follows
- **Contrapositive:** prove ¬Q → ¬P instead — logically equivalent to P → Q, per Chapter 2
- **Contradiction:** assume the claim is false, derive an impossibility, conclude the claim must be true
- √2's irrationality proof reused this chapter's own earlier "a^2 even → a even" result directly — proofs build on each other

- **Next chapter:** Mathematical induction — the one technique that can prove a property for infinitely many cases at once

CHAPTER 8 OF 10

Mathematical Induction

Discrete Mathematics Fundamentals

Chapter 8 · Mathematical Induction

Chapter 7 promised this: the one proof technique that can genuinely establish a property for infinitely many cases at once, not just the finitely many anyone happened to test. It's also the most directly programming-relevant technique in this entire course — proving a recursive function correct *is* induction, whether or not anyone names it that.

The Domino Intuition

Line up infinitely many dominoes. If you knock over the first one, and every domino is close enough to knock over the next one whenever it falls, then *all* of them fall — no matter how many there are. You never had to individually push each domino; the mechanism itself guarantees the rest.

The Formal Structure

Induction has exactly two parts:

- **Base case:** prove the property P(n) holds for the smallest value (usually n=0 or n=1) — knocking over the first domino
- **Inductive step:** assume P(k) holds for some *arbitrary* k (the "inductive hypothesis"), and prove that this forces P(k+1) to also hold — showing each domino knocks over the next

Together, using Chapter 3's own quantifier notation: `[P(0) ∧ ∀k (P(k) → P(k+1))] → ∀n P(n)`.

Worked Example: Summing the First n Integers

CLAIM: 1 + 2 + 3 + ... + N = N(N+1)/2, FOR ALL N ≥ 1

Base case (n=1): LHS = 1. RHS = 1(2)/2 = 1. Equal. ✓

Inductive step: Assume 1+2+...+k = k(k+1)/2 (the inductive hypothesis). Show it then holds for k+1:

1+2+…+k+(k+1) = [k(k+1)/2] + (k+1) *(using the inductive hypothesis to replace the first k terms)*
= (k+1)[k/2 + 1] = (k+1)(k+2)/2 — exactly the formula with n replaced by k+1. ■

Worked Example: Proving a Recursive Function Correct

This is the same technique, applied directly to code — often called **structural induction** when the "size" being inducted on is the size of a data structure.

```
def sum_list(lst):
    if len(lst) == 0:
        return 0
    return lst[0] + sum_list(lst[1:])
```

CLAIM: SUM_LIST(LST) RETURNS THE SUM OF EVERY ELEMENT IN LST, FOR ANY LIST OF LENGTH N ≥ 0

Base case (n=0): An empty list hits `len(lst) == 0` and returns 0 — the sum of no elements is 0, by convention. ✓

Inductive step: Assume `sum_list` correctly sums any list of length k (the inductive hypothesis). Consider a list of length k+1. The function computes `lst[0] + sum_list(lst[1:])`, where `lst[1:]` has length exactly k — so by the inductive hypothesis, `sum_list(lst[1:])` correctly returns the sum of the remaining k elements. Adding `lst[0]` gives the sum of all k+1 elements. ✓

By induction, `sum_list` is correct for every possible list length. ■

THIS IS EXACTLY HOW RECURSIVE FUNCTIONS GET PROVEN CORRECT

Structural induction is the formal justification behind the intuition "if the recursive call works correctly, and the base case is right, the whole function is right" — that intuition *is* mathematical induction, just applied to a function's own input size rather than to a number in a summation.

Loop Invariants Are Induction Too

THE SAME PATTERN, APPLIED TO A RUNNING LOOP

Proving a loop invariant holds after every iteration follows the identical structure: base case = the invariant holds before the loop starts (after zero iterations); inductive step = if the invariant holds before a given iteration, it still holds after that iteration runs. This is exactly how loop correctness gets formally proven — the same two-part argument, just applied to "iteration count" instead of a plain integer n.

The Common Mistake: Checking a Specific k Instead of an Arbitrary One

THE INDUCTIVE STEP HAS TO WORK FOR ANY K, NOT JUST ONE YOU HAPPENED TO CHECK

Verifying that P(1) → P(2) holds, and stopping there, is not a valid inductive step — it's just one more base case in disguise. A genuine inductive step proves the implication P(k) → P(k+1) for a completely arbitrary k, using only the assumption that P(k) holds, never a specific numeric value. This is a genuinely common mistake, and it's exactly what Exercise 3 asks you to catch.

Hands-On Exercises

EXERCISE 1

Prove by induction that $1 + 3 + 5 + \ldots + (2n - 1) = n^2$ for all $n \geq 1$ (the sum of the first n odd numbers).

View solution

EXERCISE 2

Given the recursive function `power_of_two(n)`, which returns 1 if `n == 0` and otherwise returns `2 * power_of_two(n-1)`, prove by induction that it correctly returns 2^n for every $n \geq 0$.

View solution

EXERCISE 3

Someone offers this "proof" that $3^n - 1$ is divisible by 2 for all $n \geq 1$: "Base case n=1: $3^1-1=2$, divisible by 2. Inductive step: check n=2: $3^2-1=8$, divisible by 2. Therefore, by induction, the claim holds for all n." Explain specifically why this is not a valid inductive proof, and provide a corrected inductive step.

View solution

CHAPTER 8 QUICK REFERENCE

- Induction proves a property for infinitely many cases — the domino intuition: knock over the first, guarantee each knocks over the next
- **Base case:** prove P(0) or P(1) directly — **Inductive step:** assume P(k) for arbitrary k, prove P(k+1) follows
- **Structural induction** on a recursive function's input size is exactly how recursive functions get proven correct
- Loop invariants are proven the same way — the invariant holding before an iteration implies it still holds after

- The inductive step must work for an arbitrary k — checking one specific transition (like 1→2) is not a valid inductive step, just another base case
- **Next chapter:** Combinatorics — counting, permutations, and combinations

CHAPTER 9 OF 10

Combinatorics: Counting, Permutations & Combinations

Discrete Mathematics Fundamentals

Chapter 9 · Combinatorics: Counting, Permutations & Combinations

Chapter 1 named combinatorics as underlying complexity analysis, cryptographic keyspace sizing, and test-case counting. Chapter 6 forward-referenced the pigeonhole principle. This chapter delivers the actual toolkit — starting from one foundational rule everything else in combinatorics is built from.

The Multiplication Principle

If one choice can be made in `m` ways, and a second, independent choice can be made in `n` ways, the two together can be made in `m × n` ways.

THIS IS EXACTLY WHAT NESTED LOOPS COMPUTE

A loop of size `m` nested inside a loop of size `n` executes its inner body exactly `m × n` times — the multiplication principle, running as code. Choosing among 3 colors and 4 sizes independently gives 3 × 4 = 12 total product variants, the same underlying arithmetic either way.

Factorial

`n!` ("n factorial") is the number of ways to arrange `n` distinct items in a sequence, using every item exactly once: `n! = n × (n-1) × ... × 2 × 1`. By convention, `0! = 1` — there's exactly one way to arrange zero items: the empty arrangement, doing nothing at all.

Permutations — Order Matters

`P(n, r) = n! / (n − r)!` counts the number of ways to choose *and arrange* r items from a set of n distinct items, where order matters. Ranking the top 3 finishers out of 10 racers is a permutation — 1st, 2nd, and 3rd place are genuinely different outcomes even with the same three people involved.

Combinations — Order Doesn't Matter

`C(n, r) = n! / (r! (n - r)!)`, often written "n choose r," counts the number of ways to choose r items from n where order is irrelevant. Choosing a 3-person committee from 10 people is a combination — there's no "1st, 2nd, 3rd" committee member, just membership.

THE ONE QUESTION THAT TELLS YOU WHICH FORMULA TO USE

Does the arrangement or order of the selected items actually matter for this problem? If yes — permutation. If no — combination. Nearly every mistake in this topic comes from skipping this question and guessing instead.

How the Two Formulas Relate

`C(n, r) = P(n, r) / r!` — every combination of r items corresponds to exactly r! different permutations (every possible ordering of that same selected group). Combinations divide out exactly the "redundant" orderings that permutations count separately. The two aren't unrelated formulas — combinations are permutations with the ordering information deliberately discarded.

The Pigeonhole Principle — Formalized

Chapter 6 already used this reasoning without naming it: if `n` items are placed into `m` containers and `n > m`, at least one container must hold more than one item.

THE GENERALIZED VERSION, AND WHERE CHAPTER 6'S OWN "10,000 POSSIBLE PINS" CAME FROM

The stronger, generalized form: if n items go into m containers, at least one container holds at least `⌈n/m⌉` items (rounding up). Chapter 6's own PIN exercise — 15,000 users, only 10,000 possible 4-digit PINs — concluded at least one PIN had to be shared by at least [15000/10000] = 2 users, exactly this formula in action. And that "10,000 possible PINs" figure itself comes straight from this chapter's own multiplication principle: 10 choices for each of 4 digits, with repetition allowed, gives $10 \times 10 \times 10 \times 10 = 10^4 = 10{,}000$.

Three Counting Scenarios, Side by Side

SCENARIO	FORMULA	EXAMPLE
Order matters, no repetition	P(n,r) = n!/(n−r)!	Ranking 3 medal winners from 10 racers
Order doesn't matter, no repetition	C(n,r) = n!/(r!(n−r)!)	Choosing a 3-person committee from 10 people
Order matters, repetition allowed	n^r	A 4-digit PIN, digits can repeat

Hands-On Exercises

EXERCISE 1

In how many ways can gold, silver, and bronze medals be awarded among 8 racers? Use the appropriate formula and show your work.

View solution

EXERCISE 2

How many different 3-person teams can be formed from a pool of 8 people? Compute this, and explain specifically why the answer is smaller than Exercise 1's own result, even though both start from the same 8 people and select 3.

View solution

EXERCISE 3

A room contains 32 people. Using the generalized pigeonhole principle, prove that at least 5 of them must share the same day-of-the-week birthday (Monday, Tuesday, etc. — 7 possible days).

View solution

CHAPTER 9 QUICK REFERENCE

- **Multiplication principle:** independent choices of m and n ways combine to m × n total ways — exactly what nested loops compute
- **n!** = ways to arrange n distinct items in order; 0! = 1 by convention
- **Permutation** $P(n,r) = n!/(n-r)!$ — order matters
- **Combination** $C(n,r) = n!/(r!(n-r)!)$ — order doesn't matter; $C(n,r) = P(n,r)/r!$
- The one question that decides which formula: does order/arrangement actually matter here?
- **Pigeonhole principle:** n items into m containers ($n>m$) guarantees a shared container; generalized form guarantees at least $\lceil n/m \rceil$ in some container
- **Next chapter:** Capstone — applying discrete math to real programming problems

CHAPTER 10 OF 10

Capstone: Applying Discrete Math to Real Programming Problems

Discrete Mathematics Fundamentals

Chapter 10 · Capstone — Applying Discrete Math to Real Programming Problems

One worked system, touching every chapter of this course in the order a real engineer would actually reach for each idea — designing and verifying a small access-control system for a file-sharing tool.

A Full Worked System — Designing an Access-Control Check

1 — THE RULE ITSELF, AS PROPOSITIONAL LOGIC (CH.2)

"A user can access a resource if they're an admin, or they own the resource and it isn't locked" becomes `Admin ∨ (Owner ∧ ¬Locked)`. When access is denied, a good error message needs the *negation* — applying De Morgan's Law directly: `¬(Admin ∨ (Owner ∧ ¬Locked)) ≡ ¬Admin ∧ (¬Owner ∨ Locked)`. That's precisely the reasoning a clear denial message would need: "not an admin, and either you don't own it or it's locked."

2 — THE SPECIFICATION, AS A QUANTIFIED STATEMENT (CH.3)

"Every admin can access every resource" becomes `∀u ∀r (Admin(u) → CanAccess(u, r))`. This is directly testable: `all(can_access(u, r) for u in admins for r in resources)` is the code-level check of the exact same claim.

3 — VISIBILITY, AS A SET OPERATION (CH.4)

The resources visible to a given user are `owned_by(user) ∪ (visible_to_admins if is_admin(user) else ∅)` — a direct union of two sets, exactly Chapter 4's own operation, not a special case invented separately.

4 — OWNERSHIP AND TEAMS, TWO DIFFERENT RELATIONS (CH.5)

"Owns" is a relation between users and resources — but it isn't reflexive or symmetric, and doesn't need to be; not every relation has to be an equivalence relation. "Is on the same team as," by contrast, genuinely is one — reflexive, symmetric, and transitive — and its equivalence classes are exactly the teams themselves, the same `GROUP BY` connection Chapter 5 made directly.

5 — THE USER LOOKUP, CHECKED FOR INJECTIVITY (CH.6)

The system's own `get_user(user_id)` function needs to be injective — no two different users may ever share an ID — or "the" user identified by a given ID becomes genuinely ambiguous. This isn't a stylistic preference; it's a correctness requirement for the entire system to even make sense, formalized exactly by Chapter 6's own definition.

6 — PROVING A RECURSIVE PERMISSION CHECK CORRECT, BY INDUCTION (CH.7, CH.8)

Folders can be nested — if a user isn't directly permitted on a folder, the check recurses upward to the parent. Proving this recursive check is correct for a folder tree of *any* depth is exactly structural induction: base case, the root folder (no parent, checked directly); inductive step, assuming the check is correct for a subtree of depth k, it remains correct for depth k+1, since the recursive call on the parent is covered by the inductive hypothesis. Chapter 8's own reasoning, applied directly to a real recursive function.

7 — ESTIMATING THE TEST SURFACE, BY COUNTING (CH.9)

The access rule from Step 1 combines three independent boolean conditions — Admin, Owner, Locked. By the multiplication principle, there are `2 × 2 × 2 = 8` distinct combinations to cover for full test coverage of the rule's own logic — a direct, practical answer to "how many test cases do I actually need here," not a guess.

This is, in essence, exactly what a real authorization system's design review looks like — every step traceable to a specific chapter of this course, none of it abstract math floating free of the actual code.

What This Course Doesn't Cover

In the interest of an honest accounting: **Graph Theory**, **Boolean Algebra & Digital Logic**, and **Algorithms & Complexity** were all named in Chapter 1 as deliberately out of scope, each reserved for its own future course under this same Maths for Programmers subject. Also genuinely out of scope here: formal set theory foundations (the cardinality of infinite sets, axiomatic set theory), abstract algebra (groups, rings, fields), and

formal language / automata theory. This course is the direct foundation those subjects will each build on, not a substitute for studying them when their own turn comes.

This Course's Throughline, Restated

FORMAL NOTATION IS PRECISE VOCABULARY FOR JUDGMENT YOU ALREADY MAKE

Every chapter in this course answered a version of the same question: what's the exact, formal way to say something you'd probably already reason about correctly by instinct? A programmer already knows a dict can't hold two values for one key, already knows an "if it's raining, the ground is wet" claim doesn't reverse cleanly, and already senses that testing a hundred cases isn't the same as proving something for all of them. Discrete math doesn't replace that instinct — it gives it a rigorous, checkable, communicable form, which is exactly what makes the difference between "I think this is right" and "I can show you why this is right."

Where This Course Connects

As the first course under a brand-new subject, this one has no existing site courses built on top of it yet — but it's deliberately the shared foundation for everything else this subject's own bucket list still has planned: Graph Theory will lean directly on Chapter 5's relations, Boolean Algebra & Digital Logic will lean directly on Chapter 2's propositional logic, and Algorithms & Complexity will lean directly on Chapter 9's combinatorics. Nothing here was built in isolation from where this subject is actually headed.

Hands-On Exercises

EXERCISE 1

A different system's access rule is `Admin ∧ (Owner ∨ SharedWith)`. Apply De Morgan's Law to derive the exact denial condition, showing your work step by step.

View solution

EXERCISE 2

Suppose the access rule from Step 1 grows to include a fourth independent boolean condition (say, `AccountActive`). Using this chapter's own combinatorics reasoning, how many test cases are now needed for full coverage, and why?

View solution

EXERCISE 3

For each of the seven steps in this chapter's own worked system, name the specific discrete math topic it relied on, without looking back at the step labels — just from the description of what each step actually does.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked system:** propositional logic (Ch.2) → quantified spec (Ch.3) → set operations (Ch.4) → relations (Ch.5) → function injectivity (Ch.6) → structural induction (Ch.7/8) → combinatorial test coverage (Ch.9)
- Out of scope: Graph Theory, Boolean Algebra & Digital Logic, and Algorithms & Complexity — each reserved for its own future course
- This course's throughline: formal notation gives already-good programmer instincts a rigorous, checkable form
- This course is the direct foundation this subject's own future courses will each build on
- **Course complete** — Discrete Mathematics Fundamentals, 10 chapters, from propositional logic to combinatorics