“A proof is a series of statements, each of which follows logically from what has gone before.”

— Eugenia Cheng

Session 1 of 16

PROOF100 Session 1 Introduction

AUB Math Society
Fall 2026

What a Proof Is

Daniel Dia
American University of Beirut (AUB)
https://danieldia.me/proof100/

Section 1

Course Mechanics

What is PROOF100?

PROOF100 — Foundations of Mathematical Reasoning & Computer Science

What this course is:

- A 16-session bootcamp: logic, proof, induction, sets and functions, data structures, asymptotics, computability
- You will learn to *write* proofs on paper – and to read, dissect, and debug them
- The on-ramp to **PROOF101** (Spring 2027), where a computer will check every proof you write

Who it's for:

- No prerequisites beyond high-school mathematics
- Aimed at anyone who wants rigor: math, CS, engineering, or curious

Structure & Timeline

16 sessions, September–November 2026:

S1: What a proof is (today!)
S2: Propositional logic
S3: Natural deduction
S4: Predicate logic
S5: Proof techniques
S6: Sets, relations, closures
S7: Functions, cardinality
S8: Induction I
S9: Induction II
S10: Data Structures & Algorithms I
S11: Data Structures & Algorithms II
S12: Asymptotics
S13: Recurrences, divide-and-conquer
S14: Paradigms, functional programming
S15: Computability
S16: Wrap-up

Exact dates and any schedule changes: on the course website.

The Five Problem Sets

Set	Title	Released	Due
PS1	Logic and Deduction	S3 (Mon 14 Sep)	S5 (Mon 21 Sep)
PS2	Quantifiers, Proof Techniques, Sets	S6 (Thu 24 Sep)	S8 (Thu 1 Oct)
PS3	Functions and Induction	S9 (Mon 5 Oct)	S11 (Mon 12 Oct)
PS4	Data Structures and Asymptotics	S12 (Thu 15 Oct)	S14 (Mon 26 Oct)
PS5	Recurrences, Paradigms, Computability	S15 (Mon 2 Nov)	S16 (Mon 9 Nov)

Every set contains:

- Core problems covering the three sessions it spans
- One **find-the-bug** problem: a wrong "proof" – locate the exact broken step
- One **starred problem***: harder, for those who want more

Submission: on paper, in person, at the *start* of the due session. Handwritten or typeset – illegible is neither.

Academic Integrity Policy

Collaboration:

- Discussing problems with others is allowed and *encouraged*
- Write up your own solutions in your own words
- Name anyone you worked with at the top of your submission

AI tools:

- Using an LLM to *explain a concept* is fine
- Using one to *generate a solution you submit* is not
- Any use at all must be disclosed in one line at the top of your submission

Why so strict? The skill this course builds *is* the writing. Outsource the writing and you leave with nothing.

Section 2

The High Cost of Software Bugs

The Cost of Getting It Wrong

Software bugs aren't just annoying – they're catastrophic

We'll examine three disasters that could have been prevented:

- **Ariane 5 (1996)**: $370 million rocket explosion in 37 seconds
- **Therac-25 (1985-87)**: Radiation overdoses kill 3 patients
- **Heartbleed (2014)**: 17% of the internet's encryption compromised

Common threads:

- All were created by expert teams
- All had extensive testing
- All involved seemingly "simple" bugs

Ariane 5: $370 Million in 37 Seconds

June 4, 1996: European Space Agency's Ariane 5 rocket

The Disaster:

- 37 seconds after launch, rocket self-destructed
- $370 million lost (rocket + payload)
- Years of work destroyed
- No casualties (unmanned), but a massive setback

The Cause: Integer overflow

- Reused code from Ariane 4 (cost-saving measure)
- Converted 64-bit floating-point velocity to 16-bit signed integer
- Ariane 5 was faster than Ariane 4
- Velocity exceeded 16-bit integer range ($> 32,767$)
- Overflow caused system crash

Ariane 5: The Bug

Why Testing Missed It:

- Tested with Ariane 4 flight profiles (slower velocities)
- Never tested with Ariane 5's higher velocities
- Assumed reused code was "proven" by prior flights
- Testing can only check cases you think to test

The Problematic Code (simplified):

```
// 64-bit float velocity
double vel_x = get_velocity();

// Convert to 16-bit integer
// (Range: -32768 to 32767)
int16_t vel_int = (int16_t)vel_x;

// If vel_x > 32767: OVERFLOW
```

Ariane 5: The Danger of Assumptions

The reuse fallacy: "It worked before, so it's safe"

What went wrong:

- Code was correct for Ariane 4's specifications
- Ariane 5 had different specifications (higher velocity)
- No verification that old code met new specifications
- The type system allowed an unsafe conversion

The lesson:

- Testing doesn't prove general correctness
- It proves correctness for tested scenarios only
- When requirements change, tests must change too

Past success is not a guarantee of future correctness

Therac-25: When Software Kills

1985-1987: Radiation therapy machine

The Disaster:

- 6 patients received massive radiation overdoses (100x intended)
- 3 deaths directly attributed to device
- 3 more suffered serious injuries
- One of the worst medical device failures in history

The Cause: Race condition in safety-critical code

- Machine had two modes: low-power X-ray, high-power electron beam
- Operator could edit settings rapidly
- Software had timing-dependent bug
- Fast operator inputs triggered race condition
- Safety checks bypassed, high-power beam fired without shielding

Therac-25: The Race Condition Problem

The Race Condition:

- Bug only occurred if operator edited within 8 seconds
- Required specific sequence of keystrokes
- Happened once in thousands of uses
- Intermittent failure – hardest to debug

Testing Limitations:

- Tested with “normal” operator speeds
- Didn’t test edge cases (fast editing)
- Timing-dependent bugs rarely caught in testing
- Manual testing can’t explore all timing interleavings

The fundamental challenge:

- Concurrent systems have exponentially many execution orders
- Even two operations can interleave in many ways
- Testing explores only a tiny fraction

The bug was literally waiting for the right timing to kill someone

Concurrent Systems Are Hard

The state space explosion:

- With n operations, there are $n!$ possible orderings
- 10 operations = 3.6 million possible orderings
- 20 operations = 2.4×10^{18} orderings
- Testing can check maybe thousands of orderings

Real systems are worse:

- Operations can be interrupted mid-execution
- Multiple threads running on multiple cores
- Non-deterministic timing
- Hardware reordering of operations

Testing approach: Try some orderings, hope for the best
Proving approach: Show properties hold for ALL orderings

Heartbleed: The Internet's Worst Day

April 2014: Critical vulnerability in OpenSSL

The Impact:

- Affected 17% of all secure web servers (500,000+)
- Exposed passwords, private keys, personal data
- Major sites affected: Google, Facebook, Yahoo, Amazon
- Existed for 2+ years before discovery
- Billions of dollars in damages

The Cause: Buffer over-read (bounds check missing)

- NOT a flaw in cryptography itself, but a simple programming error
- Client sends: "My payload is N bytes" + actual payload
- Server blindly trusts N without checking actual size
- **Client lies**: claims 64KB payload, sends 1 byte
- Server returns 64KB – actual payload + 64KB of memory!

Heartbleed: The Bug

Simplified version of the vulnerable code:

```
uint16_t payload_length = read_uint16(request);  // Client claims length
char* payload = read_bytes(request);

// VULNERABILITY: No check if payload_length matches reality!
char* response = malloc(payload_length);
memcpy(response, payload, payload_length);  // BUFFER OVER-READ
send_response(response, payload_length);
```

The Attack:

- Attacker sends: `payload_length = 64000`, actual payload: 1 byte
- `memcpy` copies 64000 bytes starting from payload
- Reads past end of payload into adjacent memory
- Returns 64KB of secret server memory to attacker
- Attacker retrieves passwords, encryption keys, personal data

Heartbleed: Why Testing Failed

Why Wasn't This Caught?

- OpenSSL is open source – 1000s of eyes on code
- Extensive test suite
- Used by major companies
- But: tests assumed honest clients
- Never tested malicious inputs

Testing Limitations:

- Can't test all possible inputs
- Adversarial testing requires security mindset
- Memory errors don't always crash immediately
- May pass all tests but still be exploitable

The deeper issue:

- Testers think like developers
- Attackers think differently
- Can you test for all malicious inputs?

You can't test what you don't think to test

Common Threads

- All three systems were built by **expert teams**
- All three had **extensive testing**
- All three failed on a seemingly **simple bug**

The fix, in every case, is the same:

- A *precise statement* of what must hold
- A *demonstration* that it holds – for every case, not just the tested ones

That is exactly what a proof is. This course teaches you to write them.

Section 3

The Limits of Testing

Finite Tests, Infinite Possibilities

The core limitation of testing:

- Programs have infinite possible inputs
- We can only run finite number of tests
- Bugs hide in the untested cases

Example: Simple addition function `add(x: Int64, y: Int64)`

- Possible inputs: $2^{64} \times 2^{64} = 2^{128}$ combinations
- That's roughly 3.4×10^{38} test cases
- At 1 billion tests/second: 10^{21} years to test all
- The universe is only 1.4×10^{10} years old!
- Test 1000 cases? Still leaves 10^{30+} untested

Testing gives confidence, not certainty

Dijkstra's Principle

"Testing shows the presence, not the absence of bugs" — Edsger W. Dijkstra

Categories of bugs testing fails to catch:

1. **Edge cases you didn't think of**
   - Extreme values, boundary conditions, unusual combinations
   - Example: Ariane 5 (higher velocity than tested)
2. **Rare timing-dependent bugs**
   - Race conditions, deadlocks, timing windows
   - Example: Therac-25 (8-second window)
3. **Malicious inputs**
   - Adversarial testing requires attacker mindset
   - Example: Heartbleed (lying about payload size)
4. **Complex interactions**
   - Emergent behavior from component combinations
   - Exponential state space

The Testing Paradox

What testing IS good for:

- Finding bugs during development
- Regression testing (so fixes don't break things)
- Integration testing (components work together)
- Performance testing
- User acceptance testing

What testing CANNOT guarantee:

- Absence of bugs
- Correctness for all inputs
- Freedom from race conditions

Testing is necessary but insufficient

Live Demo

- Demo 1 — binary search overflow (the JDK bug)
- Demo 2 — is $n^2 + n + 41$ always prime?
- Demo 3 — is $991n^2 + 1$ ever a perfect square?

Checking Examples $\neq$ Proving

When doing unit or integration testing, we traditionally proceed with the following cookbook:

- Come up with weird and contorted test cases
- Update the codebase, expand the programs' features, and so forth
- Test new codebase against pre-specified test cases

Remember though:

You can't test what you don't think to test

One thing you forgot to test is enough to completely compromise the security, just one!
Thus, we need another way to verify correctness.

Section 4

What is a Proof?

What is a Mathematical Statement? (1)

Definition: Statement

A **statement** is a sentence or a mathematical expression that is either definitely true or definitely false. It may be indivisible, or it may be composed of several smaller statements.

Examples:

- If a circle has radius r, then its area is πr^2 square units.
- Every even number is divisible by 2.
- $2 \in \mathbb{Z}$, $\sqrt{2} \in \mathbb{Z}$, $\mathbb{N} \subseteq \mathbb{Z}$
- The set $\{0, 1, 2\}$ has three elements.
- Some right triangles are isosceles.

Non-examples:

- Add 5 to both sides
- $\mathbb{Z}$
- 42
- What is the solution to $2x = 84$?

What is a Mathematical Statement? (2)

Definition: Predicate (Open Sentence)

A **predicate** is a sentence containing one or more variables that becomes a statement. It behaves like a "truth function" whose values depend on the input.

Examples:

- $P(x)$: "$x > 1$"
- $Q(n)$: "n is even"
- $R(x, y)$: "$x + y = 5$"
- $S(A)$: "$A \subseteq \mathbb{N}$"

Non-examples:

- $x + 1$
- "Let $a = 1$"
- "$2 > 1$"

What is a Mathematical Statement? (3)

Definition: Quantifiers

A **quantifier** turns a predicate into a statement by specifying *how many* values of the variable make it true. The **universal quantifier** $\forall$ ("for all") asserts the predicate holds for *every* value in a given set. The **existential quantifier** $\exists$ ("there exists") asserts it holds for *at least one* value.

Examples:

- $\forall n \in \mathbb{Z},\ n^2 \geq 0$ (true)
- $\exists x \in \mathbb{R},\ x^2 = 2$ (true: $x = \sqrt{2}$)
- $\forall n \in \mathbb{N},\ n$ is even (false: $n = 3$)
- $\exists n \in \mathbb{Z},\ n^2 = -1$ (false)

Watch out:

- Order matters: $\forall x\, \exists y,\ y > x$ is true, but $\exists y\, \forall x,\ y > x$ is false
- Language obscures quantifiers: "Every even number is divisible by 2" means $\forall n$
- One example is enough for $\exists$; one counterexample is enough to refute $\forall$ (De Morgan's laws)

Hypotheses and Conclusions

The two parts of a theorem

A theorem statement gives you a **hypothesis** (what you may assume) and a **conclusion** (what you must reach). The proof is the road from one to the other.

The kinds of things we prove:

- $x = y$ — something = something else
- $x \implies y$ — an implication
- $x \iff y$ — both directions
- x has some property
- $\forall x,\ p(x)$ — every x of a certain kind behaves a certain way
- $\exists x$ s.t. $p(x)$ — at least one x behaves a certain way

Hidden forms:

- "Suppose a, b, c, d are true. Then e is true" is an implication in disguise
- "Every even number is divisible by 2" hides a $\forall n$
- Same theorem, three phrasings: "If n is even, then n^2 is even" = "every even number has an even square" = "n even $\implies$ n^2 even"

Anatomy of a Proof

What a proof is

"A series of statements, each of which follows logically from what has gone before". It starts from what we assume; it ends at what we want.

Like a story – beginning, middle, end:

- **Beginning**: assumptions + the definitions of the objects involved
- **Middle**: statements, each following from what came before
- **End**: the thing we set out to prove

Steps are stepping stones across a river: spaced too far apart, you fall in.

Worked example (from the field axioms):
prove $a(b-c) = ab - ac$, given $x \cdot 0 = 0$.

$$
\begin{aligned}
a(b-c) &= a(b+(-c)) && \text{def. of } - \\
&= ab + a(-c) && \text{distributivity} \\
ac + a(-c) &= a(c+(-c)) && \text{distributivity} \\
&= a \cdot 0 = 0 && \text{inverse, given} \\
\therefore a(-c) &= -(ac) && \text{def. of inverse} \\
\therefore a(b-c) &= ab - ac && \square
\end{aligned}
$$

What Justifies a Step?

The rule

Every step must follow from what has gone before – and you must be able to say *why*.

Valid justifications:

- A **definition** (of even, of injective, ...)
- An **axiom** (e.g. the field axioms)
- A **previously proven theorem** – named, not smuggled in
- A **rule of logic** (from P and $P \implies Q$, conclude Q)
- A **hypothesis** of the theorem itself

Not justifications:

- A flying leap – skipping too many steps or using a big thm without naming
- Handwaving – vague prose instead of an argument
- A step that is simply wrong – right conclusion by wrong means proves *nothing*

How much detail? Enough for your peers to follow. In doubt: justify more, not less.

Section 5

True vs Provable

Truth is Semantic, Provability is Syntactic

Two different notions

A statement is **true** or false depending on the mathematical world it describes. A statement is **provable** when there is a chain of logical steps leading to it from accepted axioms. Truth is about *meaning*; provability is about *derivation*.

Why proofs, then?

- We have no direct access to truth – only arguments for it
- A proof is a *certificate*: checkable by anyone, step by step
- Each statement must follow logically from what has gone before – nothing else is needed to check it

The contrast with testing:

- Testing samples the world and hopes
- A proof derives the conclusion for *all* cases at once
- $991n^2 + 1$: two million true instances $\neq$ a proof; Pell's equation theory *is* one

When They Come Apart

Provability depends on the axioms

Whether a statement is provable depends on which axioms you start from. Some true statements cannot be reached from a given set of axioms at all.

The parallel postulate:

- For 2000 years, mathematicians tried to *prove* it from Euclid's other axioms
- It cannot be done: the postulate is **independent** of them
- There are geometries where it holds (the plane) and geometries where it fails (the sphere, the hyperbolic plane)
- Neither it nor its negation is provable from the remaining axioms

Gödel (1931), stated informally:

- Any consistent axiom system rich enough for arithmetic contains true statements it cannot prove
- More in S15 (Computability) – for now, know the gap exists

What "Prove" Means in This Course

The working definition

"Prove" means: derive the statement from the definitions, axioms, and results we have established – with every step justified. It does *not* mean "convince yourself it's true."

- A true statement with an unjustified argument earns no marks – the argument is the deliverable, not the verdict
- You may only use what we have defined or proven (or what the problem explicitly grants)
- Rule of thumb: justify everything enough for your peers to understand it – and when in doubt, justify more rather than less
- Problem sets grade exactly this: credit is given for naming the technique you use and identifying where each hypothesis is used
- This is exactly the standard a proof assistant will later hold you to in PROOF101: no step accepted without justification

Section 6

Set Notation as Vocabulary

The Symbols

Definition: Set (Hammack, *Book of Proof*)

"A set is a collection of things." The things in it are its **elements**; we write $x \in A$ if x is an element of A, and $x \notin A$ otherwise.

The vocabulary:

- $A \subseteq B$: every element of A is an element of B
- $A \subsetneq B$: $A \subseteq B$ and $A \neq B$
- $A \cup B = \{x : x \in A \text{ or } x \in B\}$
- $A \cap B = \{x : x \in A \text{ and } x \in B\}$
- $A \setminus B = \{x : x \in A \text{ and } x \notin B\}$
- $\emptyset = \{\}$: the set with no elements

The standard sets:

- $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ – natural numbers (**our convention**: $0 \in \mathbb{N}$; Hammack starts at 1 – read with care!)
- $\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$ – integers
- $\mathbb{Q}$ – rationals, $\mathbb{R}$ – reals
- $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$

Set-Builder Notation

The form

$\{$ expression : rule $\}$ – the set of all values of the expression, for variables satisfying the rule. Read ":" as "such that."

Examples:

- $\{x \in \mathbb{Z} : x > 0\}$ – the positive integers
- $\{2n : n \in \mathbb{Z}\}$ – the even integers
- $\{x \in \mathbb{R} : x^2 = 2\} = \{\sqrt{2}, -\sqrt{2}\}$
- $\mathbb{Q} = \left\{\frac{m}{n} : m, n \in \mathbb{Z},\ n \neq 0\right\}$

Translate both ways:

- "The odd integers"
  $\to \{2n + 1 : n \in \mathbb{Z}\}$
- $\{n \in \mathbb{N} : n \mid 12\} \to$ "the natural numbers dividing 12"
  $= \{1, 2, 3, 4, 6, 12\}$

Same set, two builders:
$\{2n : n \in \mathbb{Z}\} = \{x \in \mathbb{Z} : x \text{ is even}\}$

Common Mistakes

$\in$ is not $\subseteq$

$\in$ relates an *element* to a set; $\subseteq$ relates a *set* to a set.

Check yourself:

- $1 \in \{1, 2\}$ — true
- $1 \subseteq \{1, 2\}$ — **not even well-formed**
- $\{1\} \subseteq \{1, 2\}$ — true
- $\{1\} \in \{1, 2\}$ — false
- $\{1\} \in \{\{1\}, 2\}$ — true

The empty set:

- $\emptyset \neq \{\emptyset\}$: the first has no elements, the second has *one*
- $|\emptyset| = 0$ but $|\{\emptyset\}| = 1$
- $\emptyset \subseteq A$ for *every* set A – vacuously: there is no element of $\emptyset$ that could fail to be in A

These three confusions are where Problem Set 2 marks get lost. You have been warned.

Section 7

Writing Proofs Well

The Same Proof, Twice

Claim: If n is even, then n^2 is even.

Bad

n^2 even
$n = 2k$
$n^2 = 4k^2 = 2(2k^2)$
even ✓

- Begins at the end (the worst thing you can do)
- No words, no justification
- Where did k come from?

Good

Suppose n is even. By definition, $n = 2k$ for some $k \in \mathbb{Z}$. Then $n^2 = (2k)^2 = 4k^2 = 2(2k^2)$. Since $2k^2 \in \mathbb{Z}$, n^2 is twice an integer, so n^2 is even. □

- Assumption stated first, conclusion reached last
- Every step justified (definition, algebra)
- Full sentences: a proof is prose with symbols in it

Marking Criteria

Every problem set proof is graded on:

1. **Correctness** – the logic is valid; no gaps, no flying leaps
2. **Justification** – every nontrivial step cites a definition, a hypothesis, or a prior result
3. **Quantifier discipline** – "let x be arbitrary" for $\forall$; exhibit a witness for $\exists$; never prove $\forall$ by example
4. **Structure** – beginning (assumptions), middle (steps), end (the conclusion, *at the end*); written in sentences
5. **Notation** – symbols used correctly and defined before use

From the problem sets themselves: credit is given for stating the induction hypothesis explicitly, naming the technique you are using, and identifying where each hypothesis is used.

Our grading standard: justify everything enough for your peers to understand it. If in doubt, justify more rather than less.

Common Failure Modes

Catalogue of what a proof doesn't look like:

1. **Begin at the end, end at the beginning** – all the right steps in the wrong order prove nothing
2. **Flying leaps** – skipping steps, or using a big theorem without naming it
3. **Leaps that land in the mud** – steps that are simply wrong ($x^2 = y^2 \implies x = y$?)
4. **Handwaving** – reaching a statement by vague prose instead of logic
5. **Incorrect logic** – negating wrongly; proving the *converse* instead of the statement
6. **Incorrect assumptions or definitions** – the most avoidable error: look the definition up

And two local additions:

- **Proof by example** – one case does not prove $\forall$ (it *does* prove $\exists$)
- **"Clearly" / "obviously"** – usually marks exactly the spot where the gap is

*Every problem set contains a **find-the-bug** problem drawn from this list.*

Section 8

Summary

Session 1 Summary

The problem:

- Software bugs are catastrophically expensive (Ariane 5, Therac-25, Heartbleed)
- Testing shows the presence, not the absence of bugs – finite tests, infinite cases

The tool:

- A proof: a series of statements, each following logically from what has gone before
- Beginning (assumptions), middle (justified steps), end (the conclusion)
- One proof covers *all* cases at once – the certificate testing can never produce

The vocabulary:

- Statements, predicates, quantifiers; $\in$, $\subseteq$, $\cup$, $\cap$, $\setminus$, $\emptyset$, set-builder notation

Next session (S2): propositional logic – the machinery that makes "follows logically" precise.

Section 9

Assignments & Next Steps

This Session's Assignments

Readings (see the course website)

- Hammack, *Book of Proof* – Chapters 1
- Hammack, *Book of Proof* – Chapters 4, 5, 6

"Hand-in" Assignments (see the course website)

- Join the course WhatsApp (links on website)

Heads-up: Problem Set 1 (*Logic and Deduction*) drops at S3 (Mon 14 Sep), due at the start of S5 (Mon 21 Sep). Collaboration and AI policy apply from PS1 onward.

Questions & Discussion

Questions?

Website: https://danieldia.me/proof100/

“A proof is a series of statements, each of which follows logically from what has gone before.”

— Eugenia Cheng

Session 1 of 16

PROOF100 Session 1 Introduction

AUB Math Society
Fall 2026

What a Proof Is

Daniel Dia
American University of Beirut (AUB)
https://danieldia.me/proof100/