Applied Cryptography — Foundations

Concept cards — generated 2026-08-05

Contents

Track F

Module F01-orientation

What cryptography is trying to achieve

Before any definitions, get the goals straight. Almost every scheme in this course exists to deliver one of three things — and they are genuinely different, not shades of the same idea.

The three classical goals

Confidentiality — the adversary learns nothing about the message content. Encryption targets this.

Integrity — the message was not altered in transit. If a single bit is flipped, the receiver detects it.

Authenticity — the message really came from who it claims to. MACs and signatures target integrity and authenticity together.

The distinction is worth pausing on, because a scheme can deliver one and utterly fail the others:

> Encryption alone does **not** give you integrity. With a stream cipher, an adversary who knows the plaintext is `TRANSFER $100` can flip exactly the bits that turn it into `TRANSFER $900` — without ever decrypting anything. The message stays confidential and is now a lie.

That example is the reason the course spends real time on MACs and on authenticated encryption. "It's encrypted" is not an answer to "is it safe".

Symmetric vs. public-key

The split is about **who holds what key**.

- **Symmetric** — sender and receiver share one secret key. Fast, but you must somehow agree on that key first.

- **Public-key** (asymmetric) — a **public key** that anyone may hold, and a matching **secret key** that only its owner holds. Solves the agreement problem, at a large cost in speed.

In practice you get both: public-key crypto to agree on a symmetric key, then symmetric crypto for the actual data. That is roughly what TLS does.

Kerckhoffs's principle

> **The security of the system must rest entirely on the secrecy of the key — not on the secrecy of the design.**

Assume the adversary knows the algorithm completely. They know it is AES; they know the mode; they have read the spec. The *only* thing they do not have is your key.

This is not pessimism, it is engineering:

- Algorithms leak. They get reverse-engineered, published, or standardised.
- Keys can be changed cheaply; a design cannot.
- A design nobody may examine is a design nobody has checked. Public scrutiny is how weaknesses get found before an adversary finds them.

"Security through obscurity" names the opposite approach, and it is used as a criticism for good reason.

What to carry forward

Every security definition you meet from here will make the adversary's knowledge explicit — and it will always include the algorithm. When you read a definition and wonder "wait, is the adversary allowed to know that?", the answer is essentially always **yes, except the key**.

One boundary, stated once

CS 6260′s syllabus permits "use of AI as a study aid to better understand course material or related concepts", and prohibits it "in any form for quizzes, homework and exams" — a partially AI-derived submission carries a −15% penalty.

This trainer sits on the permitted side of that line, and it stays there because of how it is built: every problem here is an **original practice problem**, written for this app. None of them is a CS 6260 homework, quiz, or exam question. Two things follow, and they are yours to hold to:

- The handwriting reviewer on proof tasks is a **study aid on practice problems**. Pointing it at real coursework is exactly the prohibited use.
- The weekly quizzes are **closed-book and proctored**, and the exams are open only to course materials and your own notes. Nothing here is available to you during either one — which is the point. What you carry in is what you can produce unaided.

The security-game framing

This is the single most useful thing to have in your head before Lecture 1. Nearly every definition in the course is a **game**, and they all reuse the same handful of parts. Learn the pattern once and the definitions stop looking like new material.

The question a definition answers

"Is this scheme secure?" is not answerable as stated. Secure against whom, doing what, and how sure are we?

The game framing makes that precise by staging a contest:

> Someone tries to break the scheme, under stated rules, and we measure how well they do compared to trivially guessing.

The cast

Experiment (or *game*) — the whole staged contest, written as pseudocode. It says exactly what happens, in order.

Challenger — runs the experiment. It holds the secret key, makes the random choices, answers queries, and at the end decides whether the adversary won. You rarely name it explicitly; it is the code of the game itself.

Adversary (usually `A`) — the attacker. Crucially, `A` is an **arbitrary algorithm**. The definition never says how it works. It says only what `A` is allowed to *do* and what counts as winning. That is what makes a proof cover attacks nobody has invented yet.

Oracle — a black box the adversary may query. If `A` has an encryption oracle, it may hand over any message and get back a ciphertext — **without ever seeing the key**. Oracles model what an attacker can extract from the real world: if you can get a server to encrypt data of your choosing, you have an encryption oracle.

Security parameter (often `n` or `k`) — the dial for "how much security". Roughly, key length. Everything is measured as a function of it.

Advantage — the score. How much better than guessing did `A` do?

Advantage, concretely

Take a game where `A` must guess a hidden bit `b`. Guessing blind wins half the time, so raw win probability is a bad score — 1/2 means "learned nothing". Advantage subtracts that baseline:

```
Adv(A) = | Pr[A wins] - 1/2 |
```

- `A` wins 1/2 the time ⟶ advantage **0** ⟶ learned nothing.
- `A` wins always ⟶ advantage **1/2** ⟶ completely broken.
- `A` wins 0.51 of the time ⟶ advantage **0.01** ⟶ a real, if small, edge.

The absolute value matters: an adversary that is reliably *wrong* is just as informative as one reliably right — flip its answer and you have an attacker.

"Secure" then means: **every efficient adversary has negligible advantage.** Every word there is load-bearing, and F05 and F06 make *efficient* and *negligible* precise.

A worked example, in words

The IND-CPA game for encryption runs roughly like this:

1. The challenger picks a key, and secretly flips a bit `b`.
2. `A` may query an encryption oracle as much as it likes.
3. `A` submits two messages m_0, m_1 of equal length.
4. The challenger returns the encryption of `m_b` — the challenge ciphertext.
5. `A` keeps querying if it wants, then outputs a guess `b'`.
6. `A` wins if `b' = b`.

Read that as a question: *can the attacker tell which of two known messages was encrypted, even while able to encrypt anything else it likes?* If not — if every efficient `A` has negligible advantage — the scheme hides content well.

Notice what the game does **not** say: nothing about how `A` thinks. Only what it may touch, and what counts as a win.

What to carry forward

When you meet a new definition, ask these five in order:

1. What does the challenger keep secret?
2. What oracles does the adversary get?
3. What must the adversary produce?
4. What counts as winning?
5. What is the baseline the advantage subtracts?

Answer those and you have read the definition — whatever it is called.

Negligible, efficient, and PPT

These three words appear in almost every security claim in the course. They are what turn "nobody can break it" — which is false for every real scheme — into something precise and provable.

The problem they solve

Every scheme with a finite key can be broken: try all the keys. So "unbreakable" is not available. What we can say is:

> Breaking it requires **more work than anyone can do**, and even then succeeds with a probability **too small to matter**.

Efficient makes the first half precise. *Negligible* makes the second half precise.

Efficient = polynomial in the security parameter
An algorithm is **efficient** if its running time is polynomial in the security parameter n. n^2, n^5, $1000n^3$ are all efficient. 2^n is not.

PPT — *probabilistic polynomial time* — is the usual shorthand: an algorithm that may flip coins and runs in polynomial time. When a definition says "for every PPT adversary", it means every attacker you could actually build.

The subtlety that catches people, and it catches them repeatedly:

> Polynomial in **the bit-length of the input**, not in the numbers themselves.

Trial-dividing an integer N to factor it takes about $\sqrt{N}$ steps. That sounds polynomial, and it isn't — N written down is $n = \log_2 N$ bits long, so $\sqrt{N}$ is about $2^{(n/2)}$. **Exponential in the input size.** RSA rests on exactly this distinction, so it is worth getting right early. F05 drills it.

Negligible = shrinks faster than any inverse polynomial
A function $\nu(n)$ is **negligible** if it eventually falls below $1/p(n)$ for *every* polynomial p.

Informally: it goes to zero so fast that no polynomial amount of repetition rescues it.

Function	Negligible?	Why
2^{-n}	**yes**	exponential decay beats every polynomial
$2^{(-\sqrt{n})}$	**yes**	slower, still beats every polynomial
$1/n^{100}$	**no**	it *is* an inverse polynomial
$1/n^{(\log n)}$	**yes**	the exponent itself grows

$1/n^{100}$ is the instructive one. It is a fantastically small number for any real n — and still not negligible, because negligible is about the *shape of the decay*, not the size at one value of n. Run that attack n^{100} times and it succeeds; that is exactly what negligibility rules out.

Putting it together

> A scheme is **secure** if every **efficient** adversary has **negligible** advantage.

Both halves are essential. Drop *efficient* and nothing is secure — brute force always wins eventually. Drop *negligible* and you would accept schemes broken by repeating a cheap attack a polynomial number of times.

Asymptotic vs. concrete — and which CS 6260 leans on
Everything above is **asymptotic**: statements about behaviour as $n \to \infty$. It is clean, and it does not answer the question you actually have, which is *"is a 128-bit key enough for this?"*

Concrete security answers that instead, with explicit bounds:

> Any adversary running in time `t` and making `q` queries has advantage at most `q² / 2ⁿ`.

Now you can substitute your real numbers and get a real answer. This course leans concrete, so expect to plug parameters into bounds rather than only wave at limits. F06 drills both.

Bellare–Rogaway notation
The course uses the BR formalism, and the notation shows up in Lecture 1 without much ceremony. None of it is difficult — but meeting it cold, mid-definition, costs you attention you would rather spend on the idea.

Sampling

```
x ←$ S
```

"Sample `x` **uniformly at random** from the set `S`." The `$` is the whole point: it marks randomness, and it means *uniform* unless stated otherwise.

```
K ←$ {0,1}ⁿ          a uniformly random n-bit key
b ←$ {0,1}           a fair coin flip
y ←$ A(x)            run randomised algorithm A on x; y is its output
```

That last line matters: adversaries are randomised, so running one is itself a sampling step.

Plain `←` (no `$`) is ordinary assignment: `y ← f(x)` computes and stores.

Bit strings

```
{0,1}ⁿ        all bit strings of length exactly n   — there are 2ⁿ
{0,1}*        all finite bit strings, any length
|x|           the length of x in bits
x ‖ y         concatenation
x ⊕ y         bitwise XOR (same length)
ε             the empty string
```

XOR is worth a moment, because the one-time pad is the first scheme you will meet and it *is* XOR. Two facts do most of the work:

- `x ⊕ x = 0` — anything XORed with itself vanishes
- `(x ⊕ k) ⊕ k = x` — so XOR undoes itself; encryption and decryption are the same operation

Probability

```
Pr[E]                probability that event E happens
Pr[b' = b]           probability the guess matched the hidden bit
Pr[x ←$ S : P(x)]    sample x from S, then the probability P(x) holds
```

That last form — sampling on the left of the colon, the event on the right — is BR's way of putting the experiment and the question in one expression. Read the colon as "and then ask whether".

Advantage

```
Adv^{ind-cpa}_{SE}(A)
```

Read it in three parts: **superscript** is the security notion (`ind-cpa`), **subscript** is the scheme under attack (`SE`), and the **argument** is the adversary. So: "the advantage of adversary `A` against scheme `SE` in the IND-CPA sense."

Pseudocode games

Definitions are written as small procedures. A representative shape:

```
Game IND-CPA(A):
  K ←$ {0,1}ⁿ
  b ←$ {0,1}
  b' ←$ A^{Enc}(·)
  return (b' = b)

  Oracle Enc(m₀, m₁):
    return E(K, m_b)
```

Three things to read off it, in this order:

1. **`A^{Enc}`** — the superscript lists the oracles `A` may call. Here `A` gets an encryption oracle. It never gets `K`.
2. **`return (b' = b)`** — the win condition. `A` wins by guessing the hidden bit.
3. **What is never written** — how `A` works. The definition quantifies over all efficient `A`, so a proof must handle every one of them.

Common symbols

```
∀  for all          ∃  there exists       ∈  is a member of
≈  approximately    ⊆  subset of          ⌈x⌉ ⌊x⌋  ceiling, floor
negl(n)             some negligible function of n
poly(n)             some polynomial in n
⊥                   "failure" / "reject" — e.g. decryption of a bad ciphertext
```

`⊥` earns its keep once you reach MACs and authenticated encryption: it is what a verification algorithm returns when it refuses.

What to carry forward

When a game appears, read it in this order: **what is secret → what oracles the adversary gets → what counts as winning**. That is the same checklist as the security-game card, and the notation is just how it gets written down.

Module F02-discrete-math

Propositional logic, and the truth table as a tool

Later cards will hand you tables of logical equivalences. This card is about not having to trust them — because you can check any of them yourself in about thirty seconds, and knowing that changes your relationship to the whole subject.

The tool is the **truth table**. It is completely mechanical, it never requires insight, and it settles questions that otherwise feel like matters of opinion.

Propositions and connectives

A **proposition** is a statement that is definitely true or definitely false. "17 is prime" is one. "n is prime" is not — not until you say what `n` is.

We build compound propositions from `P`, `Q`, `R` with five connectives:

Symbol	Name	Read as
`¬P`	negation	not P
`P ∧ Q`	conjunction	P and Q
`P ∨ Q`	disjunction	P or Q (**inclusive**)
`P ⇒ Q`	implication	if P then Q
`P ⟺ Q`	biconditional	P if and only if Q

Building a truth table

List every combination of truth values for the variables, then evaluate. With `n` variables there are `2ⁿ` rows — 2 variables give 4 rows, 3 give 8. (That is the product rule from the sets card: one binary choice per variable.)

The five basic tables, in full:

```
 P  Q  | ¬P | P∧Q | P∨Q | P⇒Q | P⟺Q
-------+----+-----+-----+-----+-----
 F  F  | T  |  F  |  F  |  T  |  T
 F  T  | T  |  F  |  T  |  T  |  F
 T  F  | F  |  F  |  T  |  F  |  F
 T  T  | F  |  T  |  T  |  T  |  T
```

Two columns deserve a stare.

`P ∨ Q` is true when both are true. Mathematical "or" is inclusive, always, unless it explicitly says otherwise.

`P ⇒ Q` is false in exactly one row: `P` true, `Q` false. Everywhere else it is true — including both rows where `P` is false. That is the **vacuous truth**, and it is the single most counter-intuitive entry in propositional logic. "If 6 is prime then 1 = 2" is a *true statement*.

If that feels wrong, here is the reading that makes it sit right: an implication is a **promise**. "If it rains, I will bring an umbrella." You have broken that promise only if it rained and you turned up without one. If it did not rain, you kept the promise no matter what you did. The promise makes no claim about sunny days.

Logical equivalence — and how to check one

Two formulas are **logically equivalent** (`≡`) when their columns are identical in every row. Not "usually", not "for the cases I tried" — every row.

That gives you a decision procedure. **To check whether two formulas are equivalent, build both columns and compare.** There is nothing else to it.

Check 1: is `P ⇒ Q` equivalent to `¬P ∨ Q`?

```
 P  Q  | P⇒Q | ¬P | ¬PvQ
-------+-----+----+------
 F  F  |  T  | T  |  T     same
 F  T  |  T  | T  |  T     same
 T  F  |  F  | F  |  F     same
 T  T  |  T  | F  |  T     same
```

Identical in all four rows, so **yes**. This is worth knowing on its own: every implication can be rewritten as a disjunction.

Check 2: is `P ⇒ Q` equivalent to its contrapositive `¬Q ⇒ ¬P`?

```
 P  Q  | P⇒Q | ¬Q | ¬P | ¬Q⇒¬P
-------+-----+----+----+-------
 F  F  |  T  | T  | T  |   T     same
 F  T  |  T  | F  | T  |   T     same
 T  F  |  F  | T  | F  |   F     same
 T  T  |  T  | F  | F  |   T     same
```

Identical again — so **yes**, and this is not a convention to memorise. It is a verified fact, and it is *why* proving the contrapositive counts as proving the original. You have just justified an entire proof technique in four rows.

Check 3: is `P ⇒ Q` equivalent to its converse `Q ⇒ P`?

```
 P  Q  | P⇒Q | Q⇒P
-------+-----+-----
 F  F  |  T  |  T    same
 F  T  |  T  |  F    DIFFERENT
 T  F  |  F  |  T    DIFFERENT
 T  T  |  T  |  T    same
```

Two rows differ, so **no**. One differing row is enough — and notice you now have a *counterexample shape* rather than a vague sense that converses are dodgy.

De Morgan's laws

```
¬(P ∧ Q)  ≡  ¬P ∨ ¬Q
¬(P ∨ Q)  ≡  ¬P ∧ ¬Q
```

Negation flips ∧ to ∨ and pushes inward. Verify the first:

```
 P  Q  | P∧Q | ¬(P∧Q) | ¬P | ¬Q | ¬P∨¬Q
-------+-----+--------+----+----+-------
 F  F  |  F  |   T    | T  | T  |   T
 F  T  |  F  |   T    | T  | F  |   T
 T  F  |  F  |   T    | F  | T  |   T
 T  T  |  T  |   F    | F  | F  |   F
```

Identical. In words: "not both" means "at least one fails" — which is obvious once said, and the table is why you never have to wonder.

Negating an implication

Now derive the rule that costs the most marks in this course, rather than being told it.

```
¬(P ⇒ Q)  ≡  ¬(¬P ∨ Q)        by Check 1
          ≡  ¬¬P ∧ ¬Q          by De Morgan
          ≡  P ∧ ¬Q            double negation
```

> **The negation of an implication is not an implication.** It is a conjunction: `P` holds **and** `Q` fails.

This matters mechanically because **proof by contradiction starts by negating the statement**. Negate `P ⇒ Q` into `P ⇒ ¬Q` and you will spend an hour proving something you were never asked about.

Three words for whole tables

Term	Means
Tautology	true in every row (e.g. `P ∨ ¬P`)
Contradiction	false in every row (e.g. `P ∧ ¬P`)
Satisfiable	true in at least one row

"`A ≡ B`" and "`A ⟺ B` is a tautology" say the same thing.

The equivalences worth having to hand

Every one is checkable by the method above. Check a few now; trust the rest once you have.

```
¬¬P          ≡  P                      double negation
P ⇒ Q        ≡  ¬P ∨ Q                 implication as disjunction
P ⇒ Q        ≡  ¬Q ⇒ ¬P                contrapositive
¬(P ⇒ Q)     ≡  P ∧ ¬Q                 negated implication
```

```
¬(P ∧ Q)     ≡  ¬P ∨ ¬Q                  De Morgan
¬(P ∨ Q)     ≡  ¬P ∧ ¬Q                  De Morgan
P ⟺ Q        ≡  (P ⇒ Q) ∧ (Q ⇒ P)        iff is two implications
```

The last line is a working instruction: **an "if and only if" question is two proofs**, and answers that prove one direction and stop lose half the marks available.

Where the truth table runs out

Truth tables settle propositional questions completely. They cannot handle statements about *all* or *some* — `∀n. n² ≥ 0` has no finite table, because there is no finite list of rows.

That is what quantifiers are for, and F03 picks it up there. The equivalences above still hold; you just gain two more rules for moving `¬` past `∀` and `∃`. Everything in this card keeps working, and you now have a tool to check the propositional half whenever you are unsure.

Sets, relations, and the notation everything else is written in

This module exists because the proofs in F03 manipulate objects, and it is worth being fluent in those objects before being asked to reason about them. Nothing here is hard. It is vocabulary, and being shaky on vocabulary feels exactly like being bad at proofs — which it is not.

Sets

A **set** is an unordered collection of distinct things. Order and repetition carry no information:

```
{1, 2, 3}  =  {3, 1, 2}  =  {1, 1, 2, 3}
```

Notation	Read as
`x ∈ A`	`x` is an element of `A`
`x ∉ A`	`x` is not an element of `A`
`A ⊆ B`	every element of `A` is in `B`
`A ∪ B`	union — in `A`, or `B`, or both
`A ∩ B`	intersection — in both
`A \ B`	in `A` but not `B`
`∅`	the empty set
`|A|`	cardinality — how many elements

Set-builder notation describes a set by a property:

```
{ x ∈ ℤ : x is even }            the even integers
{ a ∈ ℤ_n : gcd(a, n) = 1 }      exactly the definition of Z_n*
```

Read the colon as "such that". You have already seen the second one — that *is* `ℤ_n*`, and it is nothing more than a set defined by a property.

The sets this course uses constantly

```
ℕ = {0, 1, 2, …}              naturals
ℤ = {…, −1, 0, 1, …}          integers
ℤ_n = {0, 1, …, n−1}          integers mod n
{0,1}ⁿ                        bit strings of length exactly n
{0,1}*                        bit strings of ANY finite length
```

The last two are worth pausing on, because the difference decides a definition. `{0,1}ⁿ` is finite with `2ⁿ` elements. `{0,1}*` is infinite. A hash function `H : {0,1}* → {0,1}ⁿ` maps an infinite set into a finite one — which is why collisions are *forced*, as you will see with the pigeonhole principle.

Two operations that build new sets

Cartesian product — the set of ordered pairs:

```
A × B = { (a, b) : a ∈ A, b ∈ B }        |A × B| = |A| · |B|
```

Order *does* matter inside a pair: `(1,2) ≠ (2,1)`. A pair is not a set.

Power set — the set of all subsets:

```
P(A) = { S : S ⊆ A }                     |P(A)| = 2^|A|
```

The exponent is not a coincidence: building a subset means making one independent in/out choice per element, so `|A|` binary choices give `2^|A|` outcomes. That is the product rule, and it is the same `2ⁿ` that counts `n`-bit keys.

Relations

A **relation** on a set `A` is just a set of pairs — a subset of `A × A`. Writing `a ~ b` means the pair `(a,b)` is in it. That is the whole definition; a relation is a rule for when two things are related, formalised as the list of related pairs.

Three properties matter. A relation `~` is:

Property	Means
Reflexive	`a ~ a` for every `a`
Symmetric	`a ~ b` implies `b ~ a`
Transitive	`a ~ b` and `b ~ c` implies `a ~ c`

A relation with all three is an **equivalence relation**.

Why you care: congruence is an equivalence relation

The single most important relation in this course is congruence mod `n`:

```
a ≡ b (mod n)     means      n divides (a − b)
```

Check the three properties:

- **Reflexive**: `n | (a − a) = 0`. ✓
- **Symmetric**: if `n | (a − b)` then `n | (b − a)`, since it is just the negative. ✓
- **Transitive**: if `n | (a−b)` and `n | (b−c)`, then `n` divides their sum `(a−c)`. ✓

So it is an equivalence relation, and equivalence relations **partition** the set into disjoint **equivalence classes**. Mod 7, every integer falls into exactly one of seven classes:

```
[0] = {…, −7, 0, 7, 14, …}
[1] = {…, −6, 1, 8, 15, …}
…
[6] = {…, −1, 6, 13, 20, …}
```

Seven classes, no overlaps, nothing left out. **That is what `ℤ_7` actually is** — not "the numbers 0 to 6", but the seven classes, with `0…6` as convenient representatives.

This is the justification for a move you will make constantly: replacing a number by any other number in its class. When you reduce `35 mod 7` to `0`, or swap `−1` for `6` mod 7, you are choosing a different representative of the same class. That is legal precisely *because* congruence is an equivalence relation, and the arithmetic respects the classes.

What to carry forward

You do not need to memorise this card. You need to stop pausing at `⊆`, `∈`, and `{x : P(x)}` when they appear mid-proof — because a proof is hard enough without also decoding the notation. If a symbol here is unfamiliar, that is worth ten minutes now rather than a stall in F03.

Integers: turning a definition into something you can compute with

This is the most useful card in the module, and the skill it teaches is the one that makes proofs feel possible rather than mysterious.

Here is the situation everyone gets stuck in. You are asked to prove something about an odd number. You know what "odd" means. You stare at the page. Nothing happens.

What is missing is not cleverness. It is a **translation step**, and it is mechanical.

The move: a definition is a licence to write an equation

> **"n is even"** means **there exists an integer k such that n = 2k.**

That existence claim is not decoration — it is a *thing you are allowed to write down*. The instant you assume `n` is even, you may write `n = 2k` and now you have algebra to do.

Given	Write immediately
`n` is even	`n = 2k` for some integer `k`
`n` is odd	`n = 2k + 1` for some integer `k`
`d` divides `n`	`n = dq` for some integer `q`
`a ≡ b (mod n)`	`a − b = nk` for some integer `k`
`n` is not prime (`n>1`)	`n = ab` with `1 < a, b < n`
`gcd(a,b) = d`	`d | a`, `d | b`, and `d = as + bt` (Bézout)

The whole trick: when stuck, take every hypothesis and every goal and replace each defined word with its equation. Almost always the proof is then visible, because you are looking at algebra instead of vocabulary.

Worked, slowly

Prove: if `n` is odd then `n²` is odd.

Step 1 — write what you are given, as an equation. `n` is odd, so `n = 2k + 1` for some integer `k`.

Step 2 — write what you must show, as an equation. `n²` is odd means `n² = 2m + 1` for some integer `m`. So my job is to produce such an `m`.

Step 3 — compute, aiming at that shape.

```
n² = (2k + 1)²
   = 4k² + 4k + 1
   = 2(2k² + 2k) + 1
```

Step 4 — read off the answer. Taking `m = 2k² + 2k`, which is an integer, gives `n² = 2m + 1`. So `n²` is odd. ∎

Nothing in that was inspired. Steps 1 and 2 are mechanical, step 3 is school algebra, and step 4 is noticing that step 3 already has the required shape. **Most first proofs are exactly this.**

Divisibility

`d | n` ("d divides n") means there is an integer `q` with `n = dq`.

Read `d | n` as a *statement*, true or false — not as a fraction. `3 | 12` is true; `12 | 3` is false. The bar is not division.

Properties you may use freely, each provable in one line by the translation move:

```
d | a  and  d | b     ⟹   d | (a + b)        and   d | (a − b)
d | a                 ⟹   d | ab             for any integer b
d | a  and  a | b     ⟹   d | b
```

Prove the first as an exercise in the method: `a = dq₁`, `b = dq₂`, so `a + b = d(q₁ + q₂)`, and `q₁ + q₂` is an integer. Done. That is a complete, correct proof, and it took one line.

The division algorithm

> For integers `a` and `n > 0`, there are **unique** integers `q` and `r` with `a = qn + r` and `0 ≤ r < n`.

This is what `a mod n` *is*: the remainder `r`. The two conditions matter — many pairs `(q,r)` satisfy the equation, and only one has `0 ≤ r < n`.

It also settles a common confusion: `−7 mod 3`. Uniqueness forces `−7 = (−3)(3) + 2`, so the answer is `2`, not `−1`. The remainder is never negative.

Primes

> `p > 1` is **prime** if its only positive divisors are `1` and `p`.

Note `1` is not prime — deliberately, so that factorisation into primes is unique. And the property that actually does the work in proofs:

> **Euclid's lemma.** If `p` is prime and `p | ab`, then `p | a` or `p | b`.

This fails for composites: `6 | 4·3` but `6 |/ 4` and `6 |/ 3`. Euclid's lemma is the hidden engine of the `√2` irrationality proof and of most of F07.

Greatest common divisor

> `gcd(a,b)` is the largest `d` dividing both. `a` and `b` are **coprime** when `gcd(a,b) = 1`.

The fact you will use everywhere, proved in F07:

> **Bézout.** There exist integers `s,t` with `as + bt = gcd(a,b)`.

Which is why `a` has an inverse mod `n` exactly when `gcd(a,n) = 1`: Bézout gives `as + nt = 1`, so `as ≡ 1 (mod n)`, and `s` is the inverse. The whole of `ℤ_n*` comes out of that one line.

When you are stuck, in order

1. Write what you are **given**, translating every defined word into an equation.

2. Write what you must **show**, in the same translated form.
3. Look at the gap. It is usually smaller than it felt.
4. If it is still wide, try assuming the negation of the goal (contradiction), or swapping to the contrapositive — F03 covers when each is the better bet.

Being stuck is almost never a failure of ability. It is almost always step 1 not having been done.

Sums, products, and the notation for them

`Σ` and `Π` are compressions, not concepts. Once you can expand one back into a list, they stop being obstacles — and you will meet them in every induction proof, every expectation calculation, and every security bound in the course.

Reading sigma

```
    n
    Σ  f(i)      means      f(1) + f(2) + … + f(n)
   i=1
```

Three parts: the **index** `i`, its **range** (from the bottom to the top, inclusive), and the **body** `f(i)`. Nothing else.

```
 4
 Σ  i²   =  1 + 4 + 9 + 16  =  30
i=1

 3
 Π  (i+1) =  2 · 3 · 4  =  24
i=1
```

When a Σ confuses you, expand the first three terms and the last one. That almost always makes the pattern visible, and it costs ten seconds.

The index name carries no meaning — `Σᵢ f(i)` and `Σⱼ f(j)` are the same number. It is a bound variable, exactly like the `x` in `∀x. P(x)`.

Empty and single ranges

```
 0                              1
 Σ  f(i)  =  0                  Σ  f(i)  =  f(1)
i=1                            i=1
```

An empty sum is `0`; an empty product is `1` — each is the identity of its operation, chosen so the general rules keep working. This is not pedantry: the base case of an induction is frequently an empty sum, and getting it wrong makes a correct proof look broken.

The closed forms worth knowing

Sum	Closed form
$\sum_{i=1}^{n} i$	$n(n+1)/2$
$\sum_{i=1}^{n} (2i-1)$	n^2
$\sum_{i=1}^{n} i^2$	$n(n+1)(2n+1)/6$
$\sum_{i=0}^{n} r^i$	$(r^{n+1} - 1)/(r - 1)$ for $r \neq 1$
$\sum_{i=0}^{\infty} r^i$	$1/(1 - r)$ for $\|r\| < 1$

The geometric ones matter most here. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ says that all numbers up to n bits, added together, still fall just short of one more bit — which is the counting behind bit-length arguments in F05.

The two rules you will use inside proofs

Split off the last term. This is the move that makes induction work:

$$\sum_{i=1}^{k+1} f(i) \;=\; \sum_{i=1}^{k} f(i) \;+\; f(k+1)$$

The left side is what you want at $k+1$; the right side contains exactly the thing your inductive hypothesis tells you about. **Every induction on a sum is this identity plus algebra.**

Linearity. Constants come out, and sums split:

$$\sum (a \cdot f(i) + b \cdot g(i)) \;=\; a \cdot \sum f(i) \;+\; b \cdot \sum g(i)$$

This is the same shape as linearity of expectation in F04 — and there it holds with no independence assumption, for exactly this reason: expectation is a sum, and sums split unconditionally.

Where they show up

- **Induction** — "prove $\sum_{i=1}^{n} (2i-1) = n^2$" is an F03 task, and the split-off identity is the whole inductive step.
- **Expectation** — $E[X] = \sum_x x \cdot Pr[X = x]$, a sum over outcomes.
- **Union bound** — $Pr[\bigcup A_i] \le \sum Pr[A_i]$; the right side is a Σ.
- **Birthday bound** — summing $1/N$ over $\binom{q}{2}$ pairs gives $q^2/2N$.
- **Hybrid arguments** — a chain of q steps each costing ε sums to $q \cdot \varepsilon$.

That last one is worth seeing plainly: $\sum_{i=1}^{q} \varepsilon = q \cdot \varepsilon$. The factor of q in half the bounds in this course is a sum of identical terms and nothing more mysterious than that.

Products, briefly

$\prod$ behaves the same way with multiplication. The two you will meet:

$$n! \;=\; \prod_{i}^{n} i \qquad \text{the factorial}$$

```
       i=1

φ(n) = n · Π    (1 − 1/p)      Euler's totient, over primes p dividing n
             p|n
```

The totient formula is a `Π` over a *condition* (`p | n`) rather than a numeric range. Read the subscript as "for every prime `p` dividing `n`" and expand it — for `n = 360 = 2³·3²·5` that is three factors, and the F08 task is then arithmetic.

Axioms: what you are allowed to assume

A proof derives a claim from things already accepted. So a fair question, and one almost nobody answers explicitly: **what counts as already accepted?**

Not knowing the answer produces a specific paralysis — you prove something, then worry you were supposed to prove the step before it, and the step before that. This card draws the line.

Axioms

An **axiom** is a statement assumed without proof. Every mathematical system starts with some, because otherwise justification never terminates: each fact would need an earlier fact, forever.

Axioms are not "obvious truths". They are **the starting assumptions of a system**, chosen for usefulness. Change them and you get a different, equally valid system — this is exactly what happens in F08, where the group axioms are simply declared and everything follows from them.

The ground rules for this course

Unless a problem explicitly says otherwise, you may assume without proof:

Integer arithmetic. Addition and multiplication are associative and commutative, multiplication distributes over addition, `0` and `1` are the identities, and every integer has an additive inverse. The integers are closed under `+`, `−`, `×` — sums and products of integers are integers. That closure is used constantly: `2k² + 2k` is an integer *because* `k` is.

Order. `<` is a total order compatible with the arithmetic: you may add to both sides, and multiply both sides by a positive number.

Well-ordering. Every non-empty set of non-negative integers has a least element. This looks like nothing and does real work — it is what justifies "take the smallest counterexample", and it is equivalent to induction.

The division algorithm. For `a` and `n > 0` there are unique `q, r` with `a = qn + r` and `0 ≤ r < n`. (Provable from well-ordering; cite it freely.)

Standard results already established. Once a course has proved something you may cite it by name: Bézout's identity, Euclid's lemma, unique factorisation, Lagrange's theorem, Fermat's little theorem. **Name the result you are using** — that is what makes it a citation rather than a gap.

What you may NOT assume

- **The statement you are proving.** Circular, and the most serious error you can make. It hides easily: watch for a step that quietly restates the goal.
- **Its converse.** Proving `Q ⇒ P` does not establish `P ⇒ Q`. You verified this with a truth table two cards ago.
- **That an example is a proof.** Checking `n = 1, 2, 3` establishes nothing about all `n`. It is useful for *finding* the pattern and worthless for *justifying* it.
- **Anything you have not named.** "It is well known that…" is a gap unless you say what.

Definitions are not up for debate

A definition is a naming convention, not a claim. You cannot disprove a definition; you can only apply it. When a proof asks you to show `n` is even, your obligation is fixed and finite: produce an integer `k` with `n = 2k`. That is the entire job.

This is liberating in practice. **The definition tells you exactly what you must deliver**, so "what am I even trying to do here?" always has a mechanical answer: unfold the definition of the goal and look at what it demands.

How much detail is enough?

The honest standard, and the one Bellare's writing guide uses:

> Enough that a peer who knows the definitions can follow every step without guessing.

Concretely, for this course:

Step	Justify?
`n = 2k` from "`n` is even"	state it, no justification needed
`4k² + 4k = 2(2k² + 2k)`	just show it — routine algebra
"`2k² + 2k` is an integer"	one clause: closure
"by Fermat's little theorem, `a^(p−1) ≡ 1`"	**name the theorem**
"clearly the advantage is negligible"	**not enough** — show the bound

The rule of thumb: **routine algebra can be shown, but any invoked *result* must be named.** "Clearly", "obviously", and "it is easy to see" are where marks go to die — and where the writer usually skipped the step they least understood.

Where a proof legitimately ends

A proof is finished when the goal has been derived from axioms, definitions, and cited results — and the **quantifier in the original claim has been discharged**.

That last part is the one most often left out. If the claim was "for every `n`", the proof must end by saying it holds for every `n`, not merely by finishing the algebra for an arbitrary one. It is one sentence, and it is the difference between a complete answer and one that stops just short.

How to start a proof

The hardest part of a proof is the blank page. This card is a procedure for that moment. It is deliberately mechanical, because "have an insight" is not a procedure and most proofs at this level do not require one.

The five steps

1. Write the claim out, with quantifiers explicit

Not "prove n^2 even $\Rightarrow$ n even", but:

> **Claim.** For every integer `n`, if `n²` is even then `n` is even.

You are committing to what must be shown. Half of all wrong proofs are correct proofs of a slightly different statement, and this step is where that gets caught.

2. Write GIVEN and TO SHOW, translating every definition

Draw a line down the page.

```
GIVEN                          TO SHOW
n² is even                     n is even
 → n² = 2j, some integer j      → n = 2k, some integer k
```

The arrows are the translation move from the divisibility card. **Never skip this.** The version in words is unusable; the version as equations can be manipulated.

3. Pick a technique from the shape

Look at your two columns and ask which is *concrete*.

If…	Then
GIVEN is usable	**direct** — work forward
GIVEN is awkward, negated TO SHOW is concrete	**contrapositive**
the claim says something cannot exist	**contradiction**
the objects split naturally (odd/even, ±)	**cases**
the claim is about all naturals, recursively	**induction**

In the example above, "$n^2 = 2j$" gives you almost nothing about `n`. But the *negation* of the goal — "`n` is odd" — hands you `n = 2k+1` immediately. So: contrapositive. That decision came from inspecting the columns, not from inspiration.

4. Work from both ends

Push forward from GIVEN. Separately, ask what would immediately yield TO SHOW. Then close the gap.

This is the step people skip, and it is the one that makes hard proofs tractable. Working backwards from the goal tells you what *shape* you need to produce — here, "something `= 2k + 1`" — so you know what you are steering toward instead of shuffling algebra and hoping.

5. Write it up forwards
Scratch work goes in whatever order you found it. **The final write-up runs forwards**, from hypothesis to conclusion, with each step justified.

These are two different documents. Do not hand in the first one. A proof is discovered backwards and written forwards, and conflating those is why write-ups read as chaotic even when the reasoning was sound.

The whole thing, worked

> **Claim.** For every integer `n`, if `n²` is even then `n` is even.

Steps 1–2.

```
GIVEN:    n² even  →  n² = 2j
TO SHOW:  n  even  →  n  = 2k
```

Step 3. GIVEN is awkward. Negated goal ("`n` odd") is concrete. Take the contrapositive:

> If `n` is odd then `n²` is odd.

Step 4. Forwards: `n` odd, so `n = 2k+1`, so `n² = 4k² + 4k + 1`. Backwards: to show `n²` odd I need it in the form `2m + 1`. The forward expression already is: `2(2k² + 2k) + 1`. Gap closed.

Step 5. Write up:

> **Proof.** We argue the contrapositive: we show that if `n` is odd then `n²` is odd.
>
> Suppose `n` is odd. Then `n = 2k + 1` for some integer `k`. Hence
>
> ```
> n² = (2k+1)² = 4k² + 4k + 1 = 2(2k² + 2k) + 1
> ```
>
> Since `k` is an integer, `m = 2k² + 2k` is an integer, and `n² = 2m + 1` is odd.
>
> Since a statement is equivalent to its contrapositive, the original claim follows: for every integer `n`, if `n²` is even then `n` is even. ∎

Note the first line announces the technique, and the last line discharges the quantifier. Both are cheap, and both are graded.

When you are stuck
In order, not at random:

1. **Have you translated every definition into an equation?** This is the answer about eighty percent of the time.
2. **Have you written down what you must produce?** ("An integer `k` with…")
3. **Try small cases** — `n = 1, 2, 3`. Not a proof; often shows the mechanism.
4. **Try the contrapositive.** Free to check: negate both sides and see whether the new hypothesis is more usable.
5. **Try contradiction.** Assume the negation and look for a collision.
6. **Ask what hypothesis you have not used yet.** An unused hypothesis is nearly always the missing step — problems rarely include spare ones.

The realistic expectation

Early proofs feel bad. You will stare, feel that you should already see it, and conclude that you are not a proof person.

What is actually happening is that you have not yet automated steps 1 and 2, so you are trying to reason about *words* instead of *equations*. That is genuinely hard, and it is not what a fluent reader is doing — they translated so fast it looked like insight.

Do the translation explicitly, on paper, every time. The feeling of difficulty drops sharply once it becomes automatic, and it becomes automatic through repetition rather than through talent.

Module F03-proof-technique

Propositions, quantifiers, and negation

Every security definition you will meet is a quantified statement, and most early confusion in this course is not about cryptography at all — it is about misreading a quantifier or negating an implication wrongly.

Propositions

A **proposition** is a statement that is definitely true or definitely false.

- "17 is prime" — a proposition (true)
- "n is prime" — **not** a proposition until you say what `n` is; it is a *predicate*, a statement with a free variable

Predicates become propositions when their variables are bound — by fixing a value, or by quantifying.

Connectives

Written	Means	False exactly when
`¬P`	not P	P is true
`P ∧ Q`	P and Q	either fails
`P ∨ Q`	P or Q (**inclusive**)	both fail
`P ⇒ Q`	if P then Q	**P true and Q false**
`P ⟺ Q`	P if and only if Q	they differ

Two of these trip people regularly.

"Or" is inclusive. `P ∨ Q` is true when both hold. Mathematical "or" never means "one or the other but not both" unless it says so.

An implication is only false in one situation: true hypothesis, false conclusion. So `P ⇒ Q` is *true* whenever `P` is false — the "vacuous truth". "If 6 is prime then 1 = 2" is a true statement, because the hypothesis never fires. That feels wrong until you internalise it as: *an implication promises nothing when its hypothesis fails.*

Quantifiers

```
∀x. P(x)     for every x, P(x) holds
∃x. P(x)     there exists an x with P(x)
```

Order matters, and it changes the meaning completely:

```
∀m. ∃k.  Enc(k, m) = c        for every message there is SOME key
∃k. ∀m.  Enc(k, m) = c        there is ONE key that works for every message
```

The first is weak and usually achievable; the second is a far stronger claim. In security definitions, watch for whether the adversary is chosen before or after the scheme's parameters — that ordering is exactly what a reduction has to respect.

Negation — the part that costs marks

Negation flips every quantifier and pushes inward:

Statement	Negation
`∀x. P(x)`	`∃x. ¬P(x)`
`∃x. P(x)`	`∀x. ¬P(x)`
`∀x. ∃y. P(x,y)`	`∃x. ∀y. ¬P(x,y)`
`P ∧ Q`	`¬P ∨ ¬Q`
`P ∨ Q`	`¬P ∧ ¬Q`
`P ⇒ Q`	**`P ∧ ¬Q`**

The last row is the one worth memorising. **The negation of an implication is not an implication.** It is a conjunction — a single concrete counterexample where the hypothesis holds and the conclusion fails.

Concretely: the negation of "if a scheme is secure then no efficient adversary wins" is "the scheme is secure **and** some efficient adversary wins." That is what you must exhibit to refute it.

Why this matters mechanically: **proof by contradiction begins by negating the statement.** Negate it wrongly and you spend an hour proving something else.

Converse, inverse, contrapositive
Given `P ⇒ Q`:

Name	Form	Equivalent to the original?
Converse	`Q ⇒ P`	**no**
Inverse	`¬P ⇒ ¬Q`	**no**
Contrapositive	**`¬Q ⇒ ¬P`**	**yes**

Only the contrapositive is equivalent — which is precisely why proving it counts as proving the original.

An example where the difference is obvious: "if `n` is divisible by 4 then `n` is even." True. Its converse — "if `n` is even then `n` is divisible by 4" — is false at `n = 6`. Its contrapositive — "if `n` is odd then `n` is not divisible by 4" — is true, as it must be.

Assuming a converse holds because the original does is one of the most common errors in a first proof course, and it is worth checking yourself on it deliberately.

What to carry forward
Before proving anything, write down **precisely** what you must show — with quantifiers explicit and in the right order. Then, if you plan a contradiction or a contrapositive, write the negation down too and check it against the table above. Two minutes there saves an hour of proving the wrong statement.

Definition, theorem, lemma — and what a proof is
CS 6260's stated prerequisite is "basic mathematical maturity: you have to be able to read and write mathematical definitions, statements and proofs." This card is the vocabulary that sentence assumes.

The labels
Definition — assigns precise meaning to a term. It is never true or false and is never proved; it is a naming decision. "A function `ν` is *negligible* if …" is a definition. When a proof seems impossible, check first whether you are actually being asked to *apply a definition*.

Theorem — a significant statement that has been proved. The headline result.

Lemma — a smaller proved statement, useful mainly as a step toward something larger. The distinction from theorem is about importance, not difficulty; some lemmas are harder than the theorems they serve.

Claim — a small assertion, often proved inline, used locally.

Corollary — follows quickly from a theorem just proved. "Hence, immediately…"

Proposition — a proved statement of middling importance. Between lemma and theorem, roughly.

These labels are conventions of emphasis, not a hierarchy of rigour. Everything except a definition requires proof.

What a proof has to do

> A proof is an argument that establishes a statement **beyond doubt**, from stated assumptions, by steps a reader can check.

Three parts of that are worth separating:

From stated assumptions. Everything you use is either given, previously proved, or a definition. Anything else is a gap — including things that are obviously true. If you use that `n` is an integer, that must be given.

By checkable steps. Each step follows from earlier ones for a reason the reader can verify. "It follows that" is a promise; it should be one you have kept.

Beyond doubt. Not "very likely". Not "true for the cases I tried". A million confirming examples prove nothing on their own — that is the difference between evidence and proof.

Believing vs. having proved

This is the shift the course is asking for, and it is worth naming directly.

Take: *if* n^2 *is even then* n *is even.* Test it — 4 gives 2, 16 gives 4, 100 gives 10. Every case works. You now **believe** it, entirely reasonably.

You have not proved it. You have checked finitely many of infinitely many cases. A proof must handle every integer at once, which is why it works with a *general* `n` and never with examples.

The reverse trap costs more: a statement can be true for the first thousand cases and false after. Believing without proving is how that bites you.

Reading as an adversary

Bellare's guidance, assigned by your syllabus, is to read your own writing as an enemy hunting for a flaw — someone who stops at the first thing that does not make sense. Applied to a draft proof, three questions catch most problems:

1. Is every symbol **defined** before it appears?
2. Does every operation make sense for the **type** of thing it is applied to?

3. Does each step follow, or is a word like *clearly* carrying the weight?

That is the standard the course grades against: **you are graded on what you write, not on what you meant.** The proof tasks here apply exactly those checks alongside the mathematics.

The four techniques

Your syllabus names two of these by name: "you have to know how to do proofs by contradiction and contraposition." Here is the full working set, and — more useful — how to pick between them from the *shape* of what you are asked.

Negation first

Every technique below depends on negating a statement correctly, and this is where most errors are born.

Statement	Its negation
`∀x. P(x)`	`∃x. ¬P(x)`
`∃x. P(x)`	`∀x. ¬P(x)`
`P ∧ Q`	`¬P ∨ ¬Q`
`P ∨ Q`	`¬P ∧ ¬Q`
`P ⇒ Q`	**`P ∧ ¬Q`**

The last row is the one that catches people. The negation of "if `P` then `Q`" is **not** "if `P` then not `Q`". It is "`P` holds **and** `Q` fails" — a single counterexample, with no implication left in it.

Quantifiers flip when negated. "Every key is weak" negates to "some key is not weak", not "every key is strong".

Direct proof

Assume `P`, derive `Q`.

```
To show:  if n is even then n² is even.
Proof:    n even, so n = 2k for some integer k.
          Then n² = 4k² = 2(2k²), which is even.  ∎
```

Reach for it when the hypothesis gives you something to *work with* — here, "even" hands you the factorisation immediately.

Contrapositive

`P ⇒ Q` is logically equivalent to `¬Q ⇒ ¬P`. Prove the second instead.

```
To show:  if n² is even then n is even.
Instead:  if n is odd then n² is odd.
Proof:    n odd, so n = 2k + 1.
          n² = 4k² + 4k + 1 = 2(2k² + 2k) + 1, which is odd.  ∎
```

Note *why* this is the better route: "n^2 is even" tells you almost nothing about `n` directly, while "n is odd" hands you `2k + 1` at once.

The rule of thumb: if the hypothesis is awkward to use but the *negated conclusion* is concrete, take the contrapositive. Both sides negate, and the direction reverses — get either wrong and you have proved a different statement.

Contradiction

Assume the statement is **false**, derive something impossible. Conclude the assumption was untenable.

```
To show:  √2 is irrational.
Proof:    Suppose not — √2 = a/b in lowest terms.
          … derive that a and b are both even …
          But then the fraction was not in lowest terms. Contradiction.  ∎
```

Reach for it when the statement asserts something **cannot** happen or **does not** exist. Negating gives you a concrete object to work with — here, an actual fraction with actual properties.

The discipline: state clearly what you are assuming, and at the end name exactly which two facts collide. "This is a contradiction" without saying *with what* is a gap.

Proof by cases

Split into exhaustive cases and prove each.

```
To show:  n(n+1) is even for every integer n.
Case n even:  n = 2k, so n(n+1) = 2k(n+1). Even.
Case n odd:   n + 1 is even, same argument.  ∎
```

Two obligations, and the first is the one people skip: the cases must be **exhaustive**, and each must be proved. A "proof" covering three of four cases proves nothing.

Induction

For statements about every natural number: prove `P(0)`, then prove `P(k) ⇒ P(k+1)`.

Both halves are load-bearing. Skip the base case and you can "prove" that every number is equal to every other. The inductive step must genuinely *use* the hypothesis — if it doesn't, you probably had a direct proof.

Choosing

The statement looks like…	Try
hypothesis is concrete and usable	**direct**
hypothesis is awkward; negated conclusion is concrete	**contrapositive**
asserts impossibility, non-existence, or irrationality	**contradiction**
naturally splits (odd/even, positive/negative/zero)	**cases**
about all naturals, with a recursive shape	**induction**

Choosing well is a skill this module drills directly, because on an exam nobody tells you which one to use. That is also why practice here is deliberately mixed rather than grouped by technique.

Induction — and why cryptography cares

Induction proves statements about **every** natural number using two finite pieces of work. It is standard discrete-maths material, and it earns a place here for a specific reason: **the hybrid argument, the workhorse technique for proving multi-message security, is an induction.** Getting comfortable now pays off directly when you reach reductions.

The shape

To prove `P(n)` holds for every `n ≥ 0`:

1. **Base case** — prove `P(0)`.
2. **Inductive step** — prove `P(k) ⇒ P(k+1)` for arbitrary `k`.

Then `P(n)` holds for all `n`. The image usually offered is dominoes: knock over the first, and guarantee each one topples its successor.

The assumption `P(k)` made during the step is the **inductive hypothesis**. Using it is not circular — you are not assuming what you want to prove. You are proving an *implication*, and the hypothesis is that implication's antecedent.

A worked example

Claim: `1 + 2 + ... + n = n(n+1)/2` for every `n ≥ 1`.

Base case `n = 1`: the left side is 1, the right side is `1·2/2 = 1`. ✓

Inductive step. Assume it holds for `k`:

```
1 + 2 + ... + k = k(k+1)/2                    (inductive hypothesis)
```

Add `k+1` to both sides:

```
1 + ... + k + (k+1) = k(k+1)/2 + (k+1)
                    = (k+1)(k/2 + 1)
                    = (k+1)(k+2)/2
```

which is the claim at `k+1`. ∎

Notice the step actually *used* the hypothesis. If your inductive step never invokes `P(k)`, you did not need induction — you had a direct proof.

Both obligations are load-bearing

Drop the base case and you can "prove" nonsense. Take `P(n)`: "`n = n+1`". The step goes through perfectly — assume `k = k+1`, add 1 to both sides, get `k+1 = k+2`. Every domino topples its successor. None of them ever falls, because the first was never pushed.

Drop the step and you have checked one case.

Strong induction

Sometimes `P(k)` alone is not enough and you need everything below `k`:

> Assume `P(0), P(1), …, P(k)` all hold; prove `P(k+1)`.

This is what you want for statements like "every integer `n ≥ 2` has a prime factorisation" — factoring `n` into `a·b` leaves you needing the claim for `a` and `b`, which are smaller than `n` but not necessarily `n−1`.

The payoff: hybrid arguments

Here is the connection worth carrying forward.

A security definition often guarantees something about **one** encrypted message. Real systems send many. How do you get from one to `q`?

You build a chain of `q+1` **hybrid** games:

```
H₀ :  all q messages are real
H₁ :  message 1 is random, the rest real
H₂ :  messages 1,2 random, the rest real
...
H_q:  all q messages are random
```

`H₀` is the real system; `H_q` is obviously secure — it carries no information at all. Each neighbouring pair `H_i` and `H_{i+1}` differs in **exactly one** message, so an adversary distinguishing them breaks the single-message guarantee you already have.

If no adversary can distinguish adjacent hybrids by more than `ε`, then by the triangle inequality no adversary distinguishes `H₀` from `H_q` by more than `q·ε` — and that final bound is exactly the union-bound-flavoured accumulation you saw in F04.

That chaining, step by step from a single-step guarantee to an `q`-step one, is induction. The `q·ε` is the price of the induction, and it is why security bounds in this course so often carry a factor of the number of queries.

You will meet this properly in F10. Recognising it as an old friend rather than a new trick is most of the difficulty.

Writing a proof under exam conditions

Sixty-five percent of the CS 6260 grade is timed exams. That is a different skill from doing the mathematics: the mathematics has to happen *and* be legible to a grader working quickly, under a clock, without the chance to revise. This card is about the second half.

The standard is Bellare's *Mathematical Writing*, which the syllabus assigns. Everything below is that standard applied to a timed answer.

The skeleton

Almost every proof you will be asked for fits this shape. Write the headings even when the content is short — they cost seconds and they are what a grader scans for.

```
Claim.        <restate what you are proving, with quantifiers>
Setup.        <name every object and its type>
Proof.        <the argument, one justified step per line>
Conclusion.   <discharge the original quantifier>
```

Restating the claim is not padding. It is where you commit to the right quantifier, and it is often worth a mark on its own. A proof of the wrong statement, however elegant, scores nothing.

Define before you use, and say the type

Every symbol gets introduced before it appears in an argument, with what it *is*:

> Let `A` be a PPT adversary, `k ∈ ℕ` the security parameter, `K ← {0,1}^k` uniform, and `q` the number of oracle queries `A` makes.

Type errors are the cheapest marks to lose and the easiest to avoid. An "adversary" that is used as a number, a "probability" that is a set, a `negl(·)` used without an argument — a grader reads these as not understanding the objects.

Say which technique you are using, in the first line

> "We proceed by contradiction." "We argue the contrapositive." "By induction on `q`."

One sentence, and the grader now knows what to look for and how to give partial credit. Omit it and a partially finished proof looks like a wrong one.

For reduction proofs, the four obligations

Reduction questions are the highest-value items on a CS 6260 exam and they have a fixed rubric. State all four explicitly:

1. **Construct** `B` from `A`. Say what `B` receives and what it outputs.
2. **Simulate.** Show `B` answers `A`'s oracle queries so that `A`'s view is *identical* to (or negligibly close to) the real game. This is the step most often skipped and most heavily weighted.
3. **Relate advantages.** `Adv(B) ≥ f(Adv(A))`, with the arithmetic shown.
4. **Account for resources.** `B`'s running time in terms of `A`'s, so "efficient" is preserved.

If time runs out, write the four headings with one line under each. A skeleton with the right structure earns far more than a beautifully written step 1 alone.

Under time pressure

- **Budget by marks, not by interest.** Two questions at 20 marks each beat one perfect 20 and one blank.
- **Write the claim and the technique for every part first**, then go back and fill in. This alone converts blanks into partial credit.
- **State what you are assuming when you are stuck.** "Assuming the PRF security of `F`, …" lets the grader see you know which assumption the step needs.
- **Do not erase wrong work — strike it through.** Erasing costs time, and struck work occasionally earns method marks.
- **A bound you cannot prove tightly, prove loosely.** `≤ q²/2ⁿ` when the tight answer is `q²/2ⁿ⁺¹` is worth nearly everything. Cryptography is an upper-bounding discipline; a loose bound is a real result.

Five ways answers lose marks that have nothing to do with the mathematics

Failure	What the grader sees
"It is obvious that…"	the step you were asked to prove
Symbol used before it is defined	you are not tracking your own objects
Base case omitted in an induction	half the proof is missing
Advantage claimed but never bounded	the argument does not reach a conclusion
Conclusion restates the last line, not the claim	the quantifier is never discharged

Bellare's own summary is the one to carry in: **do not make the reader guess.** Every gap you leave, the grader must fill in — and under time pressure they will not.

A worked skeleton

> **Claim.** For every PPT adversary `A` against scheme `Π`, `Adv(A)` is negligible in `k`.
>
> **Setup.** Let `A` be PPT making `q = poly(k)` queries. Let `F` be the PRF underlying `Π`, and let `ε_F(·)` bound PRF advantage.

Proof. We reduce to the PRF security of `F`. Construct `B`, which runs `A` and answers query `i` with its own oracle… `A`'s view is identical to Game 0 when `B`'s oracle is `F_K`, and to Game 1 when it is random. Hence `Adv(A) ≤ 2·ε_F(k) + q²/2ⁿ`. `B` runs in time `t_A + O(q)`.

Conclusion. `ε_F` is negligible and `q` is polynomial, so the bound is negligible; since `A` was arbitrary, the claim holds. ▮

Note the last sentence. It discharges the "for every PPT adversary" that the claim opened with. Answers routinely stop one line before this, and it is one of the cheapest marks on the paper.

Module F04-probability

Sample spaces, events, and uniform distributions

Your syllabus lists **sample space** among the terms you should already know cold. This card is that vocabulary, framed the way cryptography uses it.

The three pieces

Sample space `S` — the set of *all possible outcomes* of an experiment. Not the outcomes you care about; all of them.

Event `E ⊆ S` — a subset of the sample space. "The key is even" is an event: the set of all even keys.

Probability distribution — how likely each outcome is. Cryptography lives almost entirely on the **uniform** distribution, where every outcome is equally likely.

Under uniform, probability is pure counting:

```
Pr[E] = |E| / |S|
```

Name the sample space first

This is the habit worth building, and the one that prevents most errors.

> *A 128-bit key is chosen at random. What is the probability it starts with a zero bit?*

Sample space: `S = {0,1}¹²⁸`, so `|S| = 2¹²⁸`. Event: `E` = keys beginning with 0. The first bit is fixed and the remaining 127 are free, so `|E| = 2¹²⁷`.

```
Pr[E] = 2¹²⁷ / 2¹²⁸ = 1/2
```

Obvious in hindsight — but writing `S` down is what makes the count mechanical instead of a guess. When a probability question feels slippery, it is almost always because the sample space was never pinned down.

In the BR notation you just met

```
K ←$ {0,1}ⁿ
```

reads: the sample space is all `n`-bit strings, and the distribution is uniform. That single line is a fully specified probability experiment.

Basic rules

For events `A`, `B ⊆ S`:

```
0 ≤ Pr[A] ≤ 1
Pr[S] = 1                        something must happen
Pr[¬A] = 1 − Pr[A]               complement
Pr[A ∪ B] = Pr[A] + Pr[B] − Pr[A ∩ B]     inclusion-exclusion
Pr[A ∪ B] ≤ Pr[A] + Pr[B]                 the UNION BOUND
```

That last line is the one you will use constantly. It drops the intersection term, so it is only an upper bound — but it needs no independence and no extra information. In a security proof you usually want to say "the chance anything goes wrong is at most …", and the union bound gets you there in one step.

Why this is the language of security

Look again at what an advantage is:

```
Adv(A) = | Pr[A wins] − 1/2 |
```

`Pr[A wins]` is a probability over a sample space — the coins of the challenger *and* the coins of the adversary. When a proof says "the probability that the adversary sees a repeated nonce is at most $q^2/2^n$", it is counting a subset of that space.

Every security claim in this course is a statement about a probability. Being fluent here is not preliminary to the material; it *is* the material.

Conditional probability and independence

These two ideas are the mathematical form of the question cryptography keeps asking: **given what the adversary saw, what do they now know?**

Conditional probability

```
Pr[A | B] = Pr[A ∩ B] / Pr[B]        (defined when Pr[B] > 0)
```

Read it as: *shrink the world down to `B`, then ask how much of that smaller world is also `A`*. Under a uniform distribution this is pure counting again — count the outcomes in `B`, then count how many of those are also in `A`.

Worked example. A 3-bit key is uniform. You learn it has at least two 1-bits. What is the chance the first bit is 1?

- Outcomes with ≥ two 1s: `011, 101, 110, 111` — so `|B| = 4`
- Of those, starting with 1: `101, 110, 111` — so `|A ∩ B| = 3`

```
Pr[A | B] = 3/4
```

Unconditionally it was `1/2`. The leak moved it from 1/2 to 3/4 — and **that movement is exactly what the information was worth to an adversary.**

Independence

`A` and `B` are **independent** when knowing one tells you nothing about the other:

```
Pr[A ∩ B] = Pr[A] · Pr[B]        equivalently   Pr[A | B] = Pr[A]
```

The second form is the meaningful one: conditioning on `B` does not move the probability of `A`.

Worked example. A 4-bit key is uniform. `A` = "first bit is 1". `B` = "the key has even parity".

```
Pr[A] = 1/2      Pr[B] = 1/2      Pr[A ∩ B] = 1/4 = 1/2 · 1/2   ✓ independent
```

Knowing the parity tells you **nothing** about the first bit. That is not obvious in advance — it has to be checked — and checking is the point.

Test independence, never assume it

This is the discipline worth taking away. Independence is a *property you verify*, not a convenience you assume because two things feel unrelated.

Two traps:

Pairwise independence is not mutual independence. Three events can be independent in every pair and still be jointly dependent. Take independent fair bits `X`, `Y`, and set `Z = X ⊕ Y`. Each pair is independent — but `Z` is completely determined by the other two.

Independence is not "unrelated-looking". It is an equation. Check it.

This matters because most proof errors of this kind are silent: assuming independence lets you multiply probabilities, and the resulting bound looks perfectly reasonable while being wrong. It is also why the **union bound** is so heavily used — it needs no independence at all.

Why cryptography is written this way

Perfect secrecy is exactly an independence claim:

> The ciphertext is independent of the plaintext. `Pr[M = m | C = c] = Pr[M = m]`

Seeing the ciphertext does not move your beliefs about the message *at all*. That is the strongest possible confidentiality statement, and the one-time pad achieves it.

Every weaker definition you meet later — IND-CPA and its relatives — relaxes this into: the adversary's advantage in *distinguishing* is negligible, rather than the distributions being exactly equal. But the shape of the question never changes: **did seeing this move what the adversary knows?**

Random variables, expectation, and linearity

Your syllabus names **random variable** among the terms you must already know. Here it is, together with the one technique you will use more than any other in this course.

A random variable is a function

Not a variable, and not random — a **function from outcomes to numbers**.

```
X : S → ℝ
```

Roll two dice: the sample space is the 36 ordered pairs, and `X` = "the sum" maps each pair to a number. Draw an 8-bit key: `X` = "how many 1-bits" maps each of the 256 keys to a number in 0…8.

The randomness lives in *which outcome occurs*. `X` itself is a fixed rule.

Indicator random variables

The special case that does most of the work. An **indicator** takes only 0 or 1:

```
X_i = 1 if event A_i happens, 0 otherwise
```

Its expectation is beautifully simple:

```
E[X_i] = 1 · Pr[A_i] + 0 · Pr[¬A_i] = Pr[A_i]
```

> **The expectation of an indicator is just the probability of its event.**

That identity is the bridge between counting and probability, and it is the reason the next section works.

Expectation

```
E[X] = Σ  x · Pr[X = x]
```

The average value of `X`, weighted by how likely each value is. It need not be a value `X` can actually take — the expected number of heads in three flips is 1.5.

Linearity — the workhorse

```
E[X + Y] = E[X] + E[Y]
```

This holds always. It does not require independence. That is not a footnote; it is the whole reason the technique is powerful, because in the situations you care about — collisions, birthday bounds — the variables are *not* independent.

Worked example. An 8-bit key is uniform. What is the expected number of 1-bits?

The direct route means summing over 256 keys. Instead decompose:

```
X = X₁ + X₂ + ... + X₈        Xᵢ = 1 if bit i is a 1
E[Xᵢ] = Pr[bit i is 1] = 1/2
E[X]  = 8 × 1/2 = 4
```

Three lines, no enumeration. The pattern generalises: **write the count as a sum of indicators, take the expectation of each, add them up.**

Where this lands in cryptography

The birthday bound is exactly this argument. With `q` values drawn from a space of size `N`, let `X` count colliding pairs and put an indicator on each pair:

```
X = Σ over pairs (i,j)  X_{ij}          X_{ij} = 1 if the i-th and j-th collide
E[X_{ij}] = 1/N
E[X] = C(q,2) / N = q(q−1) / 2N
```

Notice what would have blocked a different approach: those pair-events are **not independent** — if `a` collides with `b` and `b` with `c`, then `a` collides with `c`. Linearity does not care. That is why it is the first tool to reach for.

From there, `Pr[at least one collision] ≤ E[X]` (Markov's inequality, or just the union bound over pairs), which gives the familiar `q²/2N` shape you will meet whenever the course bounds a scheme's security in terms of how many queries an adversary makes.

Counting, and the birthday bound

Almost every concrete security bound in CS 6260 is a counting argument wearing a probability hat. This card assembles the four rules you need and then derives the single most-cited number in the course: the **birthday bound**.

The four rules

Product rule. If a choice is made in `k` independent stages with `n₁, …, n_k` options, the total is `n₁ · n₂ ⋯ n_k`.

> A 128-bit key is 128 independent binary choices: `2¹²⁸` keys.

Sum rule. If a set splits into disjoint pieces, its size is the sum of the piece sizes. *Disjoint* is the load-bearing word — overlap and you double-count.

Permutations. Orderings of `n` distinct items: `n!`. Bijections from an `n`-element set to itself: also `n!`.

> `{0,1}³` has 8 elements, so `8! = 40320` permutations — the count you did earlier. A blockcipher picks one of them per key.

Combinations. Unordered `k`-subsets of `n` items:

```
C(n,k) = n! / (k!(n-k)!)
```

The one to have at your fingertips is the pair count:

```
C(n,2) = n(n-1)/2  ≈  n²/2
```

That $n^2/2$ is the whole birthday bound. Everything below is bookkeeping.

Deriving the birthday bound

Draw q values uniformly at random from a set of size N. What is the chance two of them coincide?

Counting the *ways* a collision can happen is painful. Counting the **opportunities** for one is easy, and the union bound turns opportunities into a bound:

1. There are $C(q,2) \approx q^2/2$ **pairs** of draws.
2. Any fixed pair collides with probability exactly $1/N$ — the second draw must land on whatever the first one hit.
3. The union bound (F04) says the probability that *some* pair collides is at most the sum over pairs:

$$\Pr[\text{collision}] \le C(q,2)/N \approx q^2 / 2N$$

That is it. No independence assumption was needed — which is exactly why the union bound is worth its weight: pair events here are *not* independent, and it does not care.

Reading the bound

Set the bound to a constant and solve: the probability becomes interesting when $q \approx \sqrt{N}$. Two consequences you will use constantly:

Setting	N	Collisions likely near
365 birthdays	365	$q \approx 23$
n-bit hash output	2^n	$q \approx 2^{n/2}$
128-bit block, CTR/CBC	2^{128}	$q \approx 2^{64}$ blocks

Security is halved by the birthday bound. A 256-bit hash gives 128-bit collision resistance, not 256. This is why SHA-256 is the floor for a 128-bit security target, and why 64-bit-block ciphers (3DES, Blowfish) are deprecated: 2^{32} blocks is only 32 GB, which a real connection can actually transmit.

The upper-bound habit

Notice the shape of the argument once more, because it is the shape of nearly every proof in this course:

> We could not compute the probability. We bounded it by counting the ways it could occur and summing.

You will almost never compute an adversary's success probability exactly. You will bound it. Getting comfortable with "≤" where you wanted "=" is a large part of learning to read these proofs.

Functions, bijections, and pigeonhole

Two of the course's central objects are defined by what *kind of function* they are. A blockcipher is a bijection; a hash function is deliberately not. Getting this vocabulary straight now makes the definitions of PRP, PRF and collision-resistance read as descriptions rather than jargon.

Three properties

For `f : A → B`:

Injective (one-to-one) — different inputs give different outputs. `f(x) = f(y)` forces `x = y`. Nothing collides.

Surjective (onto) — every element of `B` is hit by something.

Bijective — both. A perfect pairing between `A` and `B`, and exactly the condition for `f` to have an **inverse**.

The last point is the operational one: *a function is invertible precisely when it is a bijection.* That is not a coincidence of definitions — it is why decryption is possible.

Where each one shows up

A blockcipher is a bijection. For each key `K`, the map `E_K : {0,1}ⁿ → {0,1}ⁿ` must be a **permutation** of the block space. It has to be: decryption requires an inverse, so no two plaintexts may share a ciphertext. This is the "P" in **PRP** — pseudorandom *permutation.*

(This is also the F04 counting task you already did: there are `2ⁿ!` permutations of `{0,1}ⁿ`, and a `k`-bit key selects at most `2^k` of them. A blockcipher is a vanishingly thin slice of all permutations, and PRP security is the claim that you cannot tell the slice from the whole.)

A hash function cannot be injective. `H : {0,1}* → {0,1}ⁿ` maps inputs of *any* length into `n` bits. Infinitely many inputs, finitely many outputs. Collisions are not a flaw to be engineered away — they are forced by counting.

That is exactly why the security definition is what it is: not "collisions do not exist", which is false, but **"collisions are hard to find"**. Recognising that the definition had no other option is worth more than memorising it.

The pigeonhole principle

> If you place `n` items into `m` boxes and `n > m`, some box holds at least two items.

Trivially obvious, and surprisingly sharp. The generalised form:

> Some box holds at least `⌈n/m⌉` items.

Applied to hashing. Take `H : {0,1}⁸ → {0,1}⁴`. That is 256 possible inputs and 16 possible outputs, so:

```
⌈256 / 16⌉ = 16
```

Some output has **at least 16 distinct preimages** — regardless of how `H` is designed. You have just proved something about every possible hash function of those dimensions without knowing anything about any of them.

Notice the character of the argument: it proves collisions **exist** while giving you no way whatsoever to **find** one. That gap — existence is easy, construction is hard — is where a great deal of cryptography lives.

Why this matters for the definitions ahead

Object	Function type	Consequence
Blockcipher / PRP	bijection on `{0,1}ⁿ`	invertible; no collisions possible
PRF	arbitrary function `{0,1}ⁿ → {0,1}ᵐ`	need not be invertible
Hash	compressing, `{0,1}* → {0,1}ⁿ`	collisions guaranteed by pigeonhole
MAC	keyed, compressing	forgery, not inversion, is the threat

When you meet "PRP vs PRF" and wonder why both exist, the answer is in this table: they are different *types of function*, and a security definition can only ask for what the type permits.

Statistical distance, and the advantage it equals

An adversary's job is to tell two things apart. This card is about the number that says how well anyone possibly could — and it turns out to be a distance between distributions, computable without mentioning adversaries at all.

The definition

For distributions `P` and `Q` over a finite set `X`:

```
SD(P, Q)  =  ½ · Σ_{x ∈ X} | P(x) − Q(x) |
```

Add up how much the two disagree, point by point, and halve it.

The half is not cosmetic. Without it every difference gets counted twice — once as a surplus where `P` is larger, once as a deficit where `Q` is. The half makes `SD` land in `[0, 1]`, with 0 exactly when `P = Q` and 1 exactly when they have disjoint support.

Worked example on `{a, b, c}`:

	a	b	c
`P`	1/2	1/4	1/4
`Q`	1/4	1/4	1/2

`Σ|P−Q| = ¼ + 0 + ¼ = ½`, so `SD = ¼`.

Why it is exactly the best advantage

Let a distinguisher `D` be any function from `X` to `{0,1}` — no efficiency constraint, it may do anything. Its advantage is

```
Adv(D) = | Pr_{x←P}[D(x)=1] − Pr_{x←Q}[D(x)=1] |
```

> **The best `D` achieves exactly `SD(P, Q)`, and none does better.**

The optimal `D` is not subtle: **output 1 precisely where `P(x) > Q(x)`.** Every point where `P` exceeds `Q` adds to the gap; including any other point can only subtract. Summing the positive differences gives half the total absolute difference — which is `SD`.

Verified over 4000 random distribution pairs: the best distinguisher's advantage equals `SD(P,Q)` exactly, never off by anything.

This is the bridge the course keeps crossing. **A statement about adversaries becomes a statement about distributions**, and distributions are things you can compute with.

It is a metric — and that is what licenses hybrids

`SD` satisfies the triangle inequality:

```
SD(P, R)  ≤  SD(P, Q) + SD(Q, R)
```

Checked over 3000 random triples with no violation.

That single property is what makes a **hybrid argument** legal. To show `G₀` and `Gₙ` are indistinguishable, you build a chain `G₀, G₁, …, Gₙ` where consecutive games are close, then sum:

```
SD(G₀, Gₙ)  ≤  SD(G₀, G₁) + SD(G₁, G₂) + … + SD(Gₙ₋₁, Gₙ)
```

Each hop is small; the chain has polynomially many hops; the total is a polynomial times a negligible quantity — which the F06 lemma says is still negligible. **Every hybrid proof in the course is that sentence.**

Without the triangle inequality there would be no reason a chain of small steps adds up to a small total, and the technique would not exist.

What to carry forward

Three facts, and they are the whole toolkit:

1. `SD` is the **maximum** advantage of any distinguisher, efficient or not — so it upper-bounds what a computationally bounded adversary can do.
2. It is a **metric**, so distances chain by the triangle inequality.
3. Chaining polynomially many negligible hops leaves you negligible, by `poly × negl`.

When you meet "these two games are statistically close", it means their `SD` is negligible — and by (1) that no adversary at all, however powerful, can tell them apart with non-negligible advantage.

Module F05-asymptotics

Big-O, and what "efficient" really measures

Your syllabus is explicit: "you have to know how to measure the running time of an algorithm", and it names **big-O notation** in its list of assumed terms. The notation is the easy half. The half that actually decides whether RSA is secure is *what you measure it against.*

The notation

Written	Means	Read as
`f = O(g)`	`f` grows **no faster** than `g`	upper bound
`f = Ω(g)`	`f` grows **at least** as fast as `g`	lower bound
`f = Θ(g)`	both	tight
`f = o(g)`	`f/g → 0`	strictly slower

Constants and lower-order terms vanish: `3n² + 100n + 7 = O(n²)`. Asymptotic notation describes *shape of growth*, not size at any particular input.

A growth ranking worth knowing cold:

```
1 < log n < √n < n < n log n < n² < n³ < 2ⁿ < n!
```

The gap that matters is between `nᵏ` (**polynomial**) and `2ⁿ` (**exponential**). Everything in this course lives on one side of it or the other.

The measurement trap

Here is the point where intuition from an algorithms class quietly misleads you.

> Running time is measured in the **bit-length of the input**, not in the numerical value of the input.

An integer `N` is written with `n = ⌈log₂ N⌉ + 1` bits. So a 2048-bit number is astronomically large — about `2²⁰⁴⁸` — while its *input size* is only 2048.

Trial division. To factor `N` you test divisors up to `√N`. That looks like `O(√N)`, which sounds polynomial and is not:

```
√N  =  √(2ⁿ)  =  2^(n/2)
```

Exponential in the input size. For a 2048-bit modulus that is roughly `2¹⁰²⁴` operations — beyond any conceivable machine. This single conversion is the reason RSA can exist.

Repeated squaring. Computing `aᵉ mod N` naively means `e − 1` multiplications — again exponential in the bit-length of `e`. Square-and-multiply instead does about `log₂ e` squarings:

```
7¹²⁸ mod 13   →   128 = 2⁷   →   7 squarings, not 127 multiplications
```

That is `O(n)` for an `n`-bit exponent: **polynomial**. Which is why you can *use* RSA efficiently while an attacker cannot *break* it.

Hold those two side by side, because they are the same asymmetry:

Operation	In terms of the value	In bit-length `n`	Verdict
Trial division to `√N`	`√N`	`2^(n/2)`	exponential — infeasible
Repeated squaring	`log e`	`n`	polynomial — feasible
Brute-force `k`-bit key search	`2^k`	`2^k`	exponential — infeasible

Why the distinction is the right one

Polynomial is the accepted stand-in for "feasible", and it earns that status by being **robust**: polynomials are closed under addition, multiplication and composition. Call a polynomial-time routine a polynomial number of times and the whole thing is still polynomial. That closure is what lets a security proof compose — a reduction that runs an adversary as a subroutine stays efficient.

It is admittedly crude at the edges: `n¹⁰⁰` is polynomial and utterly impractical; `2^(n/1000)` is exponential and fine for small `n`. The definition tolerates that because the asymptotic gap is so wide in practice that the edge cases rarely arise — and where they do, **concrete security** (F06) replaces the asymptotics with explicit numbers.

Logarithms, since everything above rests on them

```
log₂ N = the number of bits needed to write N       (roughly)
log₂(2ᵏ) = k
log₂(ab) = log₂ a + log₂ b
```

If a quantity appears as `log N`, it is really "the size of `N` written down". Whenever you see `log` in a running time in this course, read it as *bit-length* and the polynomial-versus-exponential question usually answers itself.

Growth rates, logs, and reading input size correctly

F05′s other card establishes what big-O *means*. This one is about using it: the hierarchy of growth rates, the log identities you will apply without thinking, and the one substitution that decides whether an algorithm in this course counts as feasible.

The hierarchy

For large `n`, each row eventually dominates every row above it:

```
1  <  log n  <  √n  <  n  <  n log n  <  n²  <  n³  <  2ⁿ  <  n!  <  2^(n²)
```

"Eventually" is doing real work. At `n = 10`, `n²= 100` beats `2ⁿ = 1024` handily. By `n = 100` it is `10⁴` against a 31-digit number. The crossovers all happen at modest `n`, and after them the gaps are not close.

The line that matters is the one between `n^c` and `c^n`:

	polynomial `n^c`	exponential `c^n`
Double the input	cost × `2^c` — a constant factor	cost is **squared**
Add one bit	negligible change	cost × `c`
Course verdict	**efficient**	**infeasible**

Everything the course calls an "efficient adversary" lives above that line, and every hardness assumption lives below it.

Log facts worth having memorised

```
log(ab) = log a + log b          log(a/b) = log a − log b
log(aᵇ) = b · log a              log_b(x) = log₂(x) / log₂(b)
```

Two consequences used constantly:

The base only changes a constant factor. `log₂ n` and `log₁₀ n` differ by `log₂ 10 ≈ 3.32`, so inside big-O the base is invisible. Write `O(log n)` and do not worry about it.

`log` converts multiplication to addition, which is why comparing growth rates is easiest in log space. To decide whether `2⁻ⁿ` beats `n⁻¹⁰⁰`, do not evaluate either — compare `−n log 2` against `−100 log n`. That is exactly the F06 crossover task.

The substitution that decides everything

An algorithm takes an integer `N`. Its input is the *encoding* of `N`, which is

```
k = ⌈log₂(N + 1)⌉  bits
```

Complexity is measured in `k`, never in `N`. So before judging any running time, substitute `N = 2^k`:

Running time in `N`	In terms of `k = log₂ N`	Verdict
`O(log N)`	`O(k)`	efficient
`O(log² N)`	`O(k²)`	efficient
`O(√N)`	`O(2^(k/2))`	**exponential**
`O(N)`	`O(2^k)`	**exponential**

Trial division is the third row. Its `√N` looks tame and is catastrophic: each additional bit of `N` multiplies the work by `√2 ≈ 1.41`. Going from a 1024-bit to a 2048-bit modulus does not double the attacker's cost — it squares it.

This single substitution is why RSA exists. Multiplying two primes is `O(k²)`; factoring the product has no known `poly(k)` algorithm. Both are "polynomial in `N`" in some loose sense; only one is polynomial in `k`.

Bit-length arithmetic

You will do this on exams, so make it automatic.

```
bits(N) = ⌊log₂ N⌋ + 1
```

- `10¹⁰⁰` needs `⌈100 · log₂10⌉ = ⌈332.2⌉ = 333` bits.
- A 2048-bit number has about `2048 · log₁₀2 ≈ 616.5`, so **617** decimal digits.
- `n!` has about `n log₂ n − 1.44n` bits — which is why brute-forcing over permutations is hopeless well before `n = 50`.

The conversion factors worth carrying: `log₂10 ≈ 3.32` and `log₁₀2 ≈ 0.301`.

Where the constants do matter

Big-O discards constants, and occasionally you must put them back. Concrete security (F06) is exactly the discipline of not discarding them: `q²/2^(n+1)` with the `+1` intact, because the answer is a number you will act on.

Use asymptotics to decide whether an algorithm is in the feasible class at all. Use concrete bounds to choose a parameter. Applying either where the other belongs is the most common category error in the subject.

Module F06-negligibility

Negligibility and concrete security

You met "negligible" in F01 as vocabulary. Now you have the asymptotics (F05) and the probability (F04) to see what it actually says — and, just as importantly, what it refuses to say.

The definition

A function `ν : ℕ → ℝ≥0` is **negligible** if it shrinks faster than the reciprocal of *every* polynomial:

> For every polynomial `p`, there is an `N` such that `ν(n) < 1/p(n)` for all `n > N`.

The quantifier order is the whole definition. “For every polynomial” — not “for some”, not “for a large one”. You do not get to pick a polynomial and check it; the adversary picks, and you must still win.

Function	Negligible?
2^{-n}	yes
$2^{(-\sqrt{n})}$	yes
$n^{(-\log n)}$	yes
$1/n^{100}$	**no** — beaten by $p(n) = n^{101}$
$1/2^{128}$ (constant)	**no** — it is a constant, and no constant shrinks

That last row deserves a pause.

Why “negligible” is closed under the operations you need

Two facts make the definition usable, and both get used silently in proofs:

1. **Sum.** Negligible + negligible = negligible.
2. **Polynomial multiple.** $p(n) \cdot \nu(n)$ is negligible for any polynomial p.

Fact 2 is what licenses the hybrid argument. A chain of q hybrids costs you $q \cdot \varepsilon$; if q is polynomial (it is — the adversary is PPT) and ε is negligible, the total is still negligible. Without closure, every hybrid proof would collapse.

The awkward truth: asymptotics say nothing about 128

Compare 2^{-n} and n^{-100}. The first is negligible, the second is not — so asymptotically the first is the clear winner. But for actual values:

```
n = 100 :  2⁻ⁿ = 2^-100     n⁻¹⁰⁰ ≈ 2^-664.4   <- n^-100 far smaller
n = 500 :  2⁻ⁿ = 2^-500     n⁻¹⁰⁰ ≈ 2^-896.6   <- still smaller
n = 997 :  crossover        (2^-997 vs 2^-996.1)
```

Check the middle column yourself: $n^{-100} = 2^{(-100 \cdot \log_2 n)}$, and $\log_2 100 \approx 6.644$, so $100 \cdot 6.644 \approx 664.4$. Doing that arithmetic rather than taking it on trust is the habit this whole module is for.

The “non-negligible” function is *smaller* — by hundreds of orders of magnitude — everywhere you would ever deploy. Asymptotic negligibility is a statement about $n \to \infty$, and it is silent about $n = 128$.

This is not a flaw in the definition; it is the definition’s scope. It is also exactly why the course does not stop there.

Concrete security: the version with numbers in it

Bellare and Rogaway’s framing — the one CS 6260 uses — replaces “negligible” with an explicit bound:

> Any adversary running in time `t` and making `q` queries has advantage at most `f(t, q, n)`.

No limits, no asymptotics, no hidden constants. You can substitute your actual parameters and get a number.

The birthday bound is the model example. For CTR mode over an `n`-bit block cipher:

```
Adv ≤ q² / 2^(n+1)
```

Set `n = 128` and ask how many blocks you may encrypt before the advantage reaches `2⁻³²`:

```
q² / 2¹²⁹ ≤ 2⁻³²   ⟹   q² ≤ 2⁹⁷   ⟹   q ≤ 2^48.5
```

So about `2⁴⁸` blocks — roughly 4 exabytes at 16 bytes per block. That is an engineering answer: a rekeying interval you could put in a specification. "Negligible" could never have produced it.

How to hold both

	Asymptotic	Concrete
Says	secure for large enough `n`	advantage ≤ this number
Good for	clean theorems, closure, hybrids	choosing parameters
Blind to	constants and the actual `n`	nothing — but harder to prove

Read asymptotic statements as *structural* claims: this reduction exists, this composition is safe. Read concrete bounds as *engineering* claims: this key lasts this long. On an exam, "is this negligible?" is a quantifier question, and "how many queries can I make?" is an arithmetic question. Notice which one is being asked.

Module F07-modular-arithmetic

XOR algebra and the one-time pad

XOR is the most-used operation in symmetric cryptography and the least interesting one mathematically — which is exactly why it is used. This card establishes its algebra, then spends that algebra on the one-time pad, the only scheme in the course with an unconditional security proof.

The algebra

`⊕` is bitwise addition modulo 2. Four properties carry everything:

Property	Statement
Commutative	`a ⊕ b = b ⊕ a`
Associative	`(a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)`
Identity	`a ⊕ 0 = a`
Self-inverse	`a ⊕ a = 0`

The fourth is the one that matters. Every element is its own inverse, so **encryption and decryption are the same operation**:

```
c = m ⊕ k
c ⊕ k = (m ⊕ k) ⊕ k = m ⊕ (k ⊕ k) = m ⊕ 0 = m
```

Note that the derivation used associativity, then self-inverse, then identity — in that order, and nothing else. This is a group: `({0,1}ⁿ, ⊕)` is abelian, and every non-identity element has order 2. You will meet it again as a group in F08.

Uniformity: the property the OTP runs on

> **Claim.** If `K` is uniform on `{0,1}ⁿ` and independent of `M`, then `C = M ⊕ K` is uniform on `{0,1}ⁿ` and independent of `M`.

Fix any message `m` and any candidate ciphertext `c`. Then

```
Pr[C = c | M = m] = Pr[K = c ⊕ m] = 2⁻ⁿ
```

because `K` is uniform and `c ⊕ m` is one specific string. The value does not depend on `m` at all — every message maps to `c` under exactly one key, and all keys are equally likely. So the ciphertext distribution is the same whatever the message was, and observing `C` tells you nothing about `M`.

That is **perfect secrecy**, and it holds against an adversary with unbounded computing power. No assumption, no security parameter, no "efficient".

The price

The proof used `K` uniform on the *full* message space and independent of `M`. Both are expensive:

- **The key is as long as the message.** Shannon proved this is unavoidable: perfect secrecy requires `|K| ≥ |M|`. If you could shorten the key you could distinguish, because there would be ciphertexts some messages cannot produce.
- **The key is used once.** Reuse and the algebra turns on you:

```
c₁ ⊕ c₂ = (m₁ ⊕ k) ⊕ (m₂ ⊕ k) = m₁ ⊕ m₂
```

The key cancels. The adversary now holds the XOR of two plaintexts, and natural-language redundancy is usually enough to separate them. Every "two-time pad" break in history is this one line.

Why this is the pivot of the whole course

The one-time pad is perfectly secure and nearly useless. That tension is the reason everything else exists:

> Give up unconditional security, keep the key short, and settle for security against **computationally bounded** adversaries.

That trade is what a stream cipher is: replace the truly random pad with a pseudorandom one generated from a short seed. `c = m ⊕ G(k)` where `G` is a PRG. The XOR algebra is identical; only the source of the pad changed — and with it, "perfectly secret" becomes "secret unless you can distinguish `G`'s output from random".

Hold that sentence. It is the shape of nearly every argument in F10: replace a random object with a pseudorandom one, and pay for it with a distinguishing assumption.

Where you will see the algebra again

- **CBC** — `c_i = E_K(m_i ⊕ c_{i-1})`, chaining by XOR.
- **CTR** — `c_i = m_i ⊕ E_K(nonce ‖ i)`, a stream cipher built from a blockcipher. Reusing a nonce is exactly the two-time pad above.
- **Feistel networks** — the round function's output is XORed into half the block, and self-inverse is why the structure is invertible even when the round function is not.
- **GCM / polynomial MACs** — arithmetic in `GF(2^128)`, whose addition is XOR.

Modular inverses

In ordinary arithmetic, the inverse of `a` is `1/a` — the thing you multiply by to get `1`. In `Z_n` there are no fractions, so we ask the same question directly:

> The **modular inverse** of `a` mod `n` is the value `x` with `a · x ≡ 1 (mod n)`.

We write it `a⁻¹ mod n`. It is a member of `Z_n`, so it is an ordinary integer in `0 … n−1`.

When does it exist?

Not always — and the condition is the whole point:

> `a⁻¹ mod n` exists **exactly when** `gcd(a, n) = 1` (a and n are *coprime*).

Why: if some `d > 1` divides both `a` and `n`, then `a · x` is a multiple of `d` for every `x`, and so is any multiple of `n`. So `a · x` can never land on `1`, which is not a multiple of `d`. No inverse.

Finding it: extended Euclid

Ordinary Euclid gives you `gcd(a, n)`. The **extended** version also tracks *how* you got there, returning **Bezout coefficients** `x, y` with

```
a · x + n · y = gcd(a, n)
```

When `gcd(a, n) = 1` this reads `a · x + n · y = 1`. Now reduce mod `n`: the `n · y` term vanishes (it *is* a multiple of `n`), leaving

```
a · x ≡ 1 (mod n)
```

so **`x` is the inverse** — reduced into `0 … n−1`.

Worked example: `3⁻¹ mod 26`
Run Euclid on `(26, 3)`, then back-substitute:

```
26 = 8 · 3 + 2        ->   2 = 26 − 8 · 3
 3 = 1 · 2 + 1        ->   1 =  3 − 1 · 2
 2 = 2 · 1 + 0            gcd = 1, so an inverse exists
```

Back-substitute to write `1` in terms of `26` and `3`:

```
1 = 3 − 1 · 2
  = 3 − 1 · (26 − 8 · 3)
  = 3 − 26 + 8 · 3
  = 9 · 3 − 1 · 26
```

So `x = 9`, `y = −1`, and indeed `9 · 3 = 27 = 26 + 1 ≡ 1 (mod 26)`.

`3⁻¹ mod 26 = 9`.

The check that costs nothing
Always verify by multiplying back: `a · x mod n` should be `1`. If it isn't, the back-substitution slipped — that is the single most common arithmetic error in this procedure.

Modular exponentiation by repeated squaring
Every public-key operation you will meet — RSA encryption and signing, Diffie-Hellman, ElGamal — is a modular exponentiation `a^e mod n` with numbers thousands of bits long. That this is *feasible at all* rests on one algorithm, and the algorithm is a page of arithmetic.

The two rules that make it work
Reduce as you go. Modular arithmetic is compatible with multiplication:

```
(x · y) mod n  =  ((x mod n) · (y mod n)) mod n
```

So you never have to form a huge intermediate. Every partial result stays below `n`. Forming `7¹²⁸` as an integer before reducing would be legal and insane; for RSA-sized numbers it is not even storable.

Square, don't multiply. Getting from `a^k` to `a^(2k)` costs one multiplication. So the exponent doubles per step, and reaching `a^e` takes about `log₂ e` steps instead of `e`.

Square-and-multiply
Write the exponent in binary and read it left to right. Start with `1`; for each bit, **square**, and if the bit is 1, **multiply by `a`**.

Worked example, `7^13 mod 11`. Here `13 = 1101₂`.

bit	square	multiply by 7?	result mod 11
—	—	—	`1`
`1`	`1² = 1`	yes ⟶ `7`	`7`
`1`	`7² = 49 ≡ 5`	yes ⟶ `35`	`2`
`0`	`2² = 4`	no	`4`
`1`	`4² = 16 ≡ 5`	yes ⟶ `35`	`2`

So `7^13 ≡ 2 (mod 11)`. Four squarings and three multiplies, versus twelve multiplications the naive way — and the gap widens exponentially.

When the exponent is a pure power of two the multiplies vanish entirely: `7^128 mod 13` is `128 = 2⁷`, so seven squarings and nothing else.

The cost, stated properly

For a `k`-bit exponent: at most `k` squarings and at most `k` multiplies, so `O(k)` modular multiplications, each `O(k²)` with schoolbook arithmetic — `O(k³)` overall.

Put `k = 2048` in and you get a few thousand modular multiplications. A CPU does that in milliseconds.

Now put the naive method in: `2^2048` multiplications. There is no hardware, and no amount of time, that finishes it.

This is the F05 lesson in its most consequential instance. Both algorithms compute the identical value. One is polynomial in the bit-length `k`, the other exponential in `k`. The entire practicality of public-key cryptography sits in that gap.

Where it shows up

- **RSA** — `c = m^e mod N` and `m = c^d mod N`. Both are this algorithm.
- **Diffie-Hellman** — `g^a mod p`, then `(g^b)^a mod p`.
- **Fermat / Miller-Rabin primality** — computing `a^(n-1) mod n` to test `n`.
- **Fermat's little theorem inverses** — `a^(p-2) mod p` inverts `a` mod prime `p`, an alternative to extended Euclid.

The asymmetry to keep in view

Repeated squaring makes `g^a mod p` easy. Recovering `a` from `g^a mod p` — the **discrete logarithm** — has no known efficient algorithm.

That asymmetry is not incidental; it *is* the hardness assumption Diffie-Hellman stands on. A one-way function in the wild: cheap forwards, believed infeasible backwards. Notice that "believed" — it is an assumption, not a theorem, and the course will be careful about saying so.

Linear congruences: when does ax ≡ b have a solution?

Solving `ax ≡ b (mod n)` is the modular analogue of dividing. Over the rationals you divide by `a` and you are done. Modulo `n` you may get **no** solution, **one**, or **several** — and which of those happens is decided entirely by `gcd(a, n)`.

The rule

> `ax ≡ b (mod n)` has a solution **iff** `d = gcd(a, n)` divides `b`. When it does, there are exactly `d` solutions modulo `n`.

Both halves matter. The first is a solvability test you can run instantly. The second warns you that "the" solution may not be unique — an easy way to lose marks and an easy way to break a protocol.

Why it is true

`ax ≡ b (mod n)` says `ax - b = kn` for some integer `k`, i.e.

```
ax - kn = b
```

Every integer of the form `ax - kn` is a multiple of `d = gcd(a,n)`, since `d` divides both `a` and `n`. So if `d |/ b`, no `x` can work — the left side lives in `dℤ` and `b` does not.

Conversely, Bézout gives `as + nt = d`. Multiply through by `b/d` (an integer exactly when `d | b`) and you have an explicit solution. **That is why the F07 extended-Euclid task came first** — it is not a warm-up, it is the constructor.

Three cases, worked

`3x ≡ 4 (mod 7)`. `d = gcd(3,7) = 1`, which divides everything, so there is exactly one solution. `3⁻¹ ≡ 5 (mod 7)` since `15 ≡ 1`, so `x ≡ 5·4 = 20 ≡ 6`. Check: `3·6 = 18 ≡ 4 ✓`.

`14x ≡ 30 (mod 100)`. `d = gcd(14,100) = 2`, and `2 | 30`, so there are exactly **two** solutions mod 100. Divide everything through by `d`:

```
7x ≡ 15 (mod 50)
```

Now `gcd(7,50) = 1`, so invert: `7⁻¹ ≡ 43 (mod 50)`, giving `x ≡ 43·15 = 645 ≡ 45 (mod 50)`. Lift back to modulus 100 by adding multiples of 50:

```
x ≡ 45  or  x ≡ 95   (mod 100)
```

Check: `14·45 = 630 = 6·100 + 30 ✓` and `14·95 = 1330 = 13·100 + 30 ✓`.

`6x ≡ 3 (mod 9)` — wait: `d = gcd(6,9) = 3` and `3 | 3`, so three solutions exist. But **`4x ≡ 3 (mod 8)`** has `d = 4 |/ 3`: no solution at all. `4x mod 8` is only ever `0` or `4`.

The method

1. Compute `d = gcd(a, n)`.
2. If `d |/ b`, stop — **no solutions**.

3. Otherwise divide the whole congruence by `d`: `(a/d)x ≡ (b/d) (mod n/d)`.
4. Now `gcd(a/d, n/d) = 1`, so invert `a/d` by extended Euclid and solve.
5. Lift: the `d` solutions mod `n` are `x₀ + j·(n/d)` for `j = 0, …, d−1`.

Where this lands in the course
RSA key generation is a linear congruence. You choose `e` and need `d` with

```
e·d ≡ 1 (mod φ(N))
```

That is `ax ≡ b` with `b = 1`, so by the rule it is solvable **iff** `gcd(e, φ(N)) = 1` — and that is exactly the condition RSA imposes on `e`. The requirement is not a convention; it is the solvability criterion of this congruence. When you meet it in F09, you will already have proved why it must be there.

The `d = 1` case is also the one worth internalising generally: **`a` is invertible mod `n` exactly when `gcd(a, n) = 1`.** Everything about units, `Z_n*`, and Euler's totient in F08 and F09 is downstream of that sentence.

Module F08-groups

Groups: the structure underneath the arithmetic
You have been doing group theory since F07 without the vocabulary. This card supplies the vocabulary, and with it one theorem — Lagrange's — that turns a pile of modular-arithmetic facts into a single line.

The definition
A **group** is a set `G` with an operation `·` satisfying four axioms:

Axiom	Statement
Closure	`a·b ∈ G` for all `a, b ∈ G`
Associativity	`(a·b)·c = a·(b·c)`
Identity	there is `e ∈ G` with `e·a = a·e = a`
Inverses	every `a` has `a⁻¹` with `a·a⁻¹ = e`

If `a·b = b·a` as well, the group is **abelian**. Every group in this course is abelian, which is a mercy.

Note what is *not* required: no division, no ordering, no commutativity in general, and — importantly — **no second operation**. A group has one.

The three you already know
`(ℤ_n, +)` — integers mod `n` under addition. Identity `0`, inverse of `a` is `n − a`. Order `n`. Always a group, for every `n`.

`({0,1}ⁿ, ⊕)` — bit strings under XOR. Identity `0ⁿ`, and every element is its own inverse. This is the one-time pad's group from F07, and "self-inverse" is the group-theoretic statement of "encryption and decryption are the same operation".

`(ℤ_n*, ·)` — the integers mod `n` that are **invertible**, under multiplication. This one takes a moment, and it is the one that matters most.

Why `ℤ_n*` is what it is
Multiplication mod `n` is not a group on all of `ℤ_n`: `0` has no inverse, and nor does any `a` with `gcd(a, n) > 1`. So take exactly the elements that do:

```
ℤ_n* = { a ∈ ℤ_n : gcd(a, n) = 1 }
```

By the linear-congruence rule from F07, `a` is invertible mod `n` **iff** `gcd(a, n) = 1` — so this set is precisely the invertible elements, and it is a group by construction. Its size is **Euler's totient** `φ(n)`.

```
ℤ_13* = {1,…,12},          |ℤ_13*| = φ(13) = 12
ℤ_15* = {1,2,4,7,8,11,13,14}, |ℤ_15*| = φ(15) = 8
```

Closure needs one line: if `gcd(a,n) = 1` and `gcd(b,n) = 1` then `gcd(ab, n) = 1`, since a prime dividing `n` and `ab` would have to divide `a` or `b`.

Order of an element
The **order** of `a ∈ G` is the least `k ≥ 1` with `aᵏ = e`. In `ℤ_13*`:

```
2¹=2  2²=4  2³=8  2⁴=3  2⁵=6  2⁶=12  2⁷=11  2⁸=9  2⁹=5  2¹⁰=10  2¹¹=7  2¹²=1
```

Order 12 — and `2` cycles through *every* element of `ℤ_13*` on the way. An element whose order equals `|G|` is a **generator**, and a group with one is **cyclic**. `ℤ_13*` is cyclic; `2` generates it.

`ℤ_15*` is not. Its element orders are `{1, 2, 4}` and never 8, so nothing generates it. Cyclic is a real condition, not a formality — Diffie-Hellman needs a cyclic group, which is why it is set in `ℤ_p*` for prime `p`.

Lagrange's theorem, and the corollary that does the work

> **Lagrange.** In a finite group `G`, the order of every element divides `|G|`.

Look back at the orders in `ℤ_13*`: they are `1, 2, 3, 4, 6, 12` — exactly the divisors of 12. In `ℤ_15*` they are `1, 2, 4` — divisors of 8. No exceptions, because none are possible.

The corollary is the one you will use constantly:

> For every `a ∈ G`, `a^|G| = e`.

Proof: let `d` be the order of `a`. By Lagrange `d | |G|`, so `|G| = d·m`, and `a^|G| = (a^d)^m = e^m = e`. Three lines.

Now specialise. Put `G = ℤ_n*`, whose size is `φ(n)`:

```
a^φ(n) ≡ 1 (mod n)     whenever gcd(a, n) = 1
```

That is **Euler's theorem**. Put `n = p` prime, so `φ(p) = p − 1`:

```
a^(p−1) ≡ 1 (mod p)    for a not divisible by p
```

That is **Fermat's little theorem**. Both are the same corollary, read in a particular group. This is what the abstraction buys: one theorem, and the two named results of F09 fall out as instances.

What it gives you immediately

Exponents live mod `φ(n)`. Since `a^φ(n) ≡ 1`, only `e mod φ(n)` matters:

```
2^1000 mod 15 :  φ(15) = 8, and 1000 ≡ 0 (mod 8), so 2^1000 ≡ 2^0 = 1
3^100 mod 7   :  φ(7) = 6, and 100 ≡ 4 (mod 6), so 3^100 ≡ 3^4 = 81 ≡ 4
```

And RSA becomes readable. RSA needs `(m^e)^d ≡ m (mod N)`, i.e. `m^(ed) ≡ m`. Since exponents live mod `φ(N)`, it suffices that

```
e·d ≡ 1 (mod φ(N))
```

which is the linear congruence from F07 — solvable exactly when `gcd(e, φ(N)) = 1`. Every piece of RSA key generation is now a consequence of things you have proved, rather than a recipe to memorise.

Cyclic groups and generators

Lagrange told you the order of an element divides `|G|`. This card is about the elements where that division is trivial — where one element's powers sweep out the *entire* group. Those elements are what public-key cryptography is built on, and the group where finding them is easy but reversing them is hard is exactly where Diffie–Hellman and ElGamal live.

Generators

An element `g ∈ G` is a **generator** when its order equals `|G|`. Equivalently, every element of `G` is `g^k` for some `k`:

```
G = { g^1, g^2, g^3, …, g^|G| }
```

A group with a generator is **cyclic**. Not every group is.

Take $\mathbb{Z}_{13}^*$, which has 12 elements. The powers of 2 are

```
2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1
```

Twelve distinct values — all of $\mathbb{Z}_{13}^*$. So 2 is a generator and $\mathbb{Z}_{13}^*$ is cyclic.

Now $\mathbb{Z}_{15}^*$, which has 8 elements. Every element has order 1, 2 or 4 — **none has order 8**. So $\mathbb{Z}_{15}^*$ is not cyclic and has no generator at all.

Group	|G|	Element orders present	Cyclic?
$\mathbb{Z}_{13}^*$	12	1, 2, 3, 4, 6, 12	yes
$\mathbb{Z}_{7}^*$	6	1, 2, 3, 6	yes
$\mathbb{Z}_{15}^*$	8	1, 2, 4	**no**
$\mathbb{Z}_{12}^*$	4	1, 2	**no**

Read the pattern off the table: a group is cyclic exactly when some element's order reaches `|G|`. Lagrange guarantees orders *divide* `|G|`; it does not promise any element achieves it.

Which groups are cyclic
The fact the course uses constantly:

> **`ℤ_p*` is cyclic for every prime `p`.**

That is why cryptographic constructions specify a prime modulus rather than any modulus — it guarantees a generator exists. For composite `n`, `ℤ_n*` is usually not cyclic, as $\mathbb{Z}_{15}^*$ shows.

How many generators
If `G` is cyclic with `|G| = m`, it has exactly `φ(m)` generators.

For $\mathbb{Z}_{13}^*$: `m = 12`, and `φ(12) = 4`. The generators are 2, 6, 7 and 11 — four of them, as promised. So generators are common enough to find by guessing and checking, which is exactly how real key generation does it.

Why this is the load-bearing structure
Fix a generator `g` of `ℤ_p*`. Then every element is `g^k`, so the map

```
k  ↦  g^k  (mod p)
```

is a bijection from exponents onto the group. Computing it is cheap — that is square-and-multiply from F07. **Inverting it is the discrete logarithm problem, and no efficient algorithm is known.**

That asymmetry is not a detail of one scheme. It is the trapdoor underneath Diffie–Hellman key exchange, ElGamal encryption, DSA signatures and Schnorr signatures. Every one of them is "exponentiation is easy, taking logs is hard", running in a cyclic group you chose precisely because it has a generator.

What to carry forward
When a scheme says "let `g` be a generator of a group of prime order `q`", it is invoking three things at once: a generator exists (the group is cyclic), its powers cover everything (so ciphertexts range over the whole group), and the log is hard (so the exponent stays secret). Recognising that phrase as a package rather than as three unrelated words is most of what reading these definitions requires.

Module F09-number-theory

The Chinese Remainder Theorem

Two congruences with coprime moduli pin down exactly one value. That is the whole statement, and it is the reason RSA decryption runs about four times faster than it looks like it should.

The statement

Let n_1, n_2 be **coprime**. Then for any a_1, a_2 the system

$$
\begin{aligned}
x &\equiv a_1 \pmod{n_1} \\
x &\equiv a_2 \pmod{n_2}
\end{aligned}
$$

has a solution, and it is **unique modulo $n_1 n_2$**.

Coprimality is not decoration. Drop it and both halves fail: the system can have no solution at all, or many.

A worked example

$$
\begin{aligned}
x &\equiv 2 \pmod{3} \\
x &\equiv 3 \pmod{5}
\end{aligned}
$$

$\gcd(3,5) = 1$, so expect one answer mod 15. Walk the first congruence: $x \in \{2, 5, 8, 11, 14\}$. Of those, which is $3 \bmod 5$? Only **8**.

Check: $8 = 2 \cdot 3 + 2$ ✓ and $8 = 1 \cdot 5 + 3$ ✓. And 8 is the only solution below 15.

Why "unique mod $n_1 n_2$" is the interesting half

Existence you might guess. Uniqueness says something stronger: the map

$$x \mapsto (x \bmod n_1,\ x \bmod n_2)$$

is a **bijection** from $\mathbb{Z}_{n_1 n_2}$ onto $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2}$. Counting confirms it — both sides have $n_1 n_2$ elements, and CRT says nothing collides.

So an integer mod $n_1 n_2$ and its pair of residues are two encodings of the same thing. You may move between them freely, and arithmetic works componentwise: add the pairs, multiply the pairs, and convert back.

Where the course uses it

RSA decryption. The modulus is $N = pq$ with p, q distinct primes — coprime by construction. Instead of computing $c^d \bmod N$ with a 2048-bit modulus, compute it twice with 1024-bit moduli:

$$
\begin{aligned}
m_1 &= c^{d \bmod p-1} \pmod{p} \\
m_2 &= c^{d \bmod q-1} \pmod{q}
\end{aligned}
$$

then recombine by CRT. Modular exponentiation costs roughly the cube of the modulus length, so halving the length cuts each of the two exponentiations to about an eighth — a net speedup near **4×**. Real RSA implementations all do this.

It is also how `φ(N) = (p−1)(q−1)` is derived, and it is why leaking one of the two CRT halves during decryption breaks the key — a fault attack you will meet when the course covers RSA in practice.

What to carry forward

When you see a composite modulus that factors into coprime pieces, the reflex is: **split it**. The problem becomes independent smaller problems, and the recombination is guaranteed to work and to be unique. That is CRT being useful rather than CRT being stated.

Quadratic residues and Euler's criterion

Half the nonzero elements mod a prime are squares. Deciding which half is easy; finding the square root modulo a composite is not. That gap is a second trapdoor alongside discrete log, and it is where Rabin encryption lives.

The definition

`a` is a **quadratic residue** mod `p` when some `x` satisfies `x² ≡ a (mod p)`. Otherwise it is a non-residue.

Mod 11, square everything and collect the results:

```
1²=1   2²=4   3²=9   4²=5   5²=3
6²=3   7²=5   8²=9   9²=4  10²=1
```

The residues are `{1, 3, 4, 5, 9}` — five of the ten nonzero elements.

Exactly half, always

For odd prime `p` there are exactly `(p−1)/2` quadratic residues.

The reason is visible in the table above: `x` and `p−x` square to the same value, so squaring is **two-to-one** on the nonzero elements. Half as many outputs as inputs.

`p`	nonzero elements	residues	count
11	10	{1,3,4,5,9}	5
13	12	{1,3,4,9,10,12}	6
17	16	{1,2,4,8,9,13,15,16}	8

Euler's criterion — deciding without searching

You do not have to try every `x`. For odd prime `p` and `a` not divisible by `p`:

```
a^((p-1)/2)  ≡  +1  (mod p)    if a is a quadratic residue
             ≡  −1  (mod p)    if it is not
```

Check it mod 11, where `(p−1)/2 = 5`:

- `3` is a residue: `3⁵ = 243 = 22·11 + 1 ≡ 1` ✓
- `2` is not: `2⁵ = 32 = 2·11 + 10 ≡ 10 ≡ −1` ✓

That is one modular exponentiation — cheap, by square-and-multiply from F07. So **testing** whether a square root exists is easy.

Why it works: Fermat gives `a^(p−1) ≡ 1`, so `a^((p−1)/2)` squares to 1, so it is `+1` or `−1` — the only square roots of 1 modulo a prime. Which one it is tracks exactly whether `a` is itself a square.

The asymmetry that matters

- **Deciding** whether `a` is a residue mod prime `p`: easy, by the criterion.
- **Computing** the square root mod prime `p`: also easy, given an algorithm.
- **Either one modulo `N = pq` with `p, q` secret**: believed hard, and provably **as hard as factoring `N`**.

That last line is the cryptographic content. Rabin encryption rests on it, and it is the cleanest example in the course of a problem whose difficulty is *equivalent to* factoring rather than merely related to it.

What to carry forward

Euler's criterion is a pattern worth recognising on sight: an exponentiation that returns `±1` and answers a yes/no question about structure. When a scheme computes `a^((p−1)/2)`, it is asking "is this a square?" — and knowing that saves you decoding the algebra every time it appears.

The discrete logarithm problem

Every asymmetric scheme in the second half of the course rests on one asymmetry: in a cyclic group, exponentiating is cheap and undoing it is not. This card is about the "not".

The problem

Fix a cyclic group `G` of order `q` with generator `g`. Because `g` generates, every element is `g^k` for exactly one `k ∈ {0, …, q−1}`. So

```
dlog_g : G → ℤ_q ,    g^k ↦ k
```

is a well-defined bijection. The **discrete logarithm problem** is: given `h`, find `k` with `g^k = h`.

Small example in $\mathbb{Z}_{13}^*$ with `g = 2`:

```
k:     1   2   3   4   5   6   7   8   9  10  11  12
2^k:   2   4   8   3   6  12  11   9   5  10   7   1
```

$\text{dlog}_2(11) = 7$, because $2^7 = 128 = 9 \cdot 13 + 11$.

Read the second row again. It is a *permutation* of `{1,…,12}` in scrambled order — no pattern to exploit, which is the intuition for why inverting it is hard.

The asymmetry, quantified

Direction	Cost
`k ↦ g^k`	about `2 log₂ k` multiplications — square-and-multiply from F07
`g^k ↦ k`	best known general algorithm about `√q` operations

For a 256-bit prime order `q`, forward is a few hundred multiplications and backward is about `2¹²⁸` operations. **The gap is the entire trapdoor.**

Note `√q`, not `q`. Generic algorithms (baby-step giant-step, Pollard's rho) do better than brute force, which is exactly why a 256-bit group is chosen to buy 128-bit security — the same square-root rule you met in the birthday bound, for the same combinatorial reason.

Where the group choice matters

Discrete log is hard **in some groups and easy in others**, and the difficulty is a property of the group rather than of the notation:

- `ℤ_p*` for a large prime `p`: believed hard.
- Elliptic curve groups: believed harder per bit, which is why they use smaller parameters for the same security.
- `ℤ_n` under **addition**: trivially easy. `g^k` is just `k·g mod n`, so recovering `k` is one modular division.

That last row is worth holding onto. "Discrete log is hard" is not a statement about exponent notation — it is a claim about a specific group, and writing an additive group multiplicatively does not make anything hard.

What it buys

Three constructions you will meet, all the same trapdoor:

- **Diffie–Hellman**: Alice sends `g^a`, Bob sends `g^b`, both compute `g^(ab)`. An eavesdropper sees `g^a` and `g^b` and would need a discrete log to proceed.
- **ElGamal encryption**: the ciphertext hides the message behind `g^(ab)`.
- **DSA / Schnorr signatures**: the private key is an exponent; the public key is `g` raised to it.

What to carry forward

When a scheme says "let `g` generate a group of prime order `q`", it is importing this whole card: a generator exists, exponentiation covers the group, the log is hard, and `q` prime keeps the structure clean. Recognising the phrase as that package is most of what reading these definitions requires.

Primality testing, and why passing a test is not a proof

RSA and Diffie–Hellman both start by generating a large prime. Nobody factors a 2048-bit number to check — the whole point is that factoring is hard. So how do you know it is prime?

You do not, exactly. You test, and the tests are probabilistic. Understanding what they do and do not establish is the content of this card.

The Fermat test
Fermat says: if `p` is prime and `gcd(a, p) = 1` then `a^(p−1) ≡ 1 (mod p)`.

Run it backwards as a test. Given `n`, pick a base `a` and compute `a^(n−1) mod n`:

- Result ≠ **1** ⟶ `n` is **definitely composite**. The contrapositive of a theorem, so this half is certain.
- Result = **1** ⟶ `n` is *probably* prime. This half is not certain at all.

One modular exponentiation, cheap by square-and-multiply. That asymmetry — certain when it fails, uncertain when it passes — is typical of the whole family.

Where it breaks: Carmichael numbers
Take `n = 561 = 3 · 11 · 17`. Plainly composite. Test base 2:

```
2^560 mod 561 = 1
```

The test says "probably prime". It is wrong.

And this is not bad luck with one base. **Every** `a` coprime to 561 gives 1 — verified by exhaustive check. Those are the **Carmichael numbers**, and 561 is the smallest. There are infinitely many.

So the Fermat test cannot be rescued by trying more bases. Against a Carmichael number every coprime base lies, and the only bases that expose it are those sharing a factor with `n` — which you would find by gcd anyway, without any exponentiation.

Miller–Rabin: the fix
Miller–Rabin exploits something Fermat throws away. For odd `n`, write `n − 1 = 2^s · d` with `d` odd, then look at the whole squaring chain

```
a^d,  a^(2d),  a^(4d),  …,  a^(2^s·d) = a^(n−1)
```

Modulo a **prime**, the only square roots of 1 are `+1` and `−1`. So if the chain reaches 1, the value just before it must have been `+1` or `−1`. If the chain reaches 1 from anything else, you have found a square root of 1 that is neither — and `n` cannot be prime.

Run it on 561 with base 2. Here `560 = 2⁴ · 35`:

```
2^35   ≡ 263
2^70   ≡ 166
2^140  ≡ 67
2^280  ≡ 1     ← reached 1 from 67, which is neither +1 nor 560
```

Composite, detected. The Fermat test saw only the final 1 and was fooled; Miller–Rabin saw how it got there.

Each round with a random base catches a composite with probability at least 3/4, so `k` rounds leave error at most `4^(-k)`. Forty rounds is far below the chance of a hardware fault, which is why implementations call this "probably prime" and ship it.

What to carry forward

Two habits, both transferable well past primality:

1. **A test that passes proves less than a test that fails.** Failure invokes the contrapositive of a theorem and is certain; passing is consistent with a counterexample you have not found.
2. **When a test is fooled, ask what it discarded.** Fermat looks only at the endpoint; Miller–Rabin looks at the path. The extra information was available the whole time.

The same shape recurs when you meet security proofs: an adversary that fails tells you little, and a definition that only checks the final output can miss an attack that is visible in how the output was reached.

Prime-order groups, safe primes, and extracting square roots

Every discrete-log scheme opens with a phrase like "let `g` generate a group of prime order `q`". This card is about why that sentence is so specific, and how such a group is actually built.

Why prime order

Take a cyclic group `G` of order `q` with `q` **prime**. Lagrange says every element's order divides `q`. The only divisors are 1 and `q`, so every element is either the identity or has order exactly `q`.

> **In a group of prime order, every non-identity element is a generator.**

Check it in the additive group of order 5, 7, 11 or 13: the number of generators equals the number of non-identity elements, every time.

Three consequences that matter:

- **No small subgroups.** In `ℤ_p*` with `p` prime, `|G| = p−1` is even and composite, so subgroups of order 2, 3, … exist. An attacker who tricks you into working in one has reduced the discrete-log problem to a tiny group. That is the **small-subgroup attack**, and prime order removes the target.
- **Picking a generator is free.** Choose anything but the identity.
- **Exponents live mod `q`**, a prime, so every nonzero exponent is invertible — which signature schemes rely on.

Building one: safe primes

`ℤ_p*` has order `p−1`, which is never prime for `p > 3`. So you carve out a prime-order subgroup instead.

A **safe prime** is `p = 2q + 1` where `q` is also prime — 11, 23, 47, 59, 83, 107, 167 and 179 all qualify.

Then `|ℤ_p*| = p − 1 = 2q`, whose only divisors are 1, 2, `q`, `2q`. So `ℤ_p*` has exactly one subgroup of order `q`: **the squares**. Take any `h` and set `g = h²`; unless `g = 1`, it generates a group of prime order `q`.

With `p = 23` and `q = 11`, the squares are

```
{1, 2, 3, 4, 6, 8, 9, 12, 13, 16, 18}
```

— exactly 11 elements, as promised.

This is why real parameter sets ship a safe prime rather than any prime. The structure is the security property.

Extracting square roots when `p ≡ 3 (mod 4)`

Euler's criterion tells you *whether* a square root exists. Finding it is a separate problem, and for a large class of primes it is a one-liner.

If `p ≡ 3 (mod 4)` and `a` is a quadratic residue mod `p`:

```
x = a^((p+1)/4)  (mod p)
```

is a square root of `a`. Verified exhaustively for `p = 7, 11, 19, 23, 31, 43` — it works for every residue.

Why: `x² = a^((p+1)/2) = a · a^((p−1)/2) = a · 1 = a`, using Euler's criterion for the last step. The exponent is an integer precisely because `p + 1` is divisible by 4, which is what `p ≡ 3 (mod 4)` buys.

Example: `p = 11`, `a = 5`. Then `(p+1)/4 = 3` and `5³ = 125 ≡ 4`, and indeed `4² = 16 ≡ 5 (mod 11)`.

For `p ≡ 1 (mod 4)` no such formula exists and you need the Tonelli–Shanks algorithm — which is one reason implementations prefer `p ≡ 3 (mod 4)`.

What to carry forward

Parameter choices in real schemes are load-bearing, not conventional. "Prime order" blocks small-subgroup attacks. "Safe prime" is how prime order is obtained. "`p ≡ 3 mod 4`" makes square roots computable in one exponentiation. When a specification pins a parameter down that precisely, the reason is usually an attack it is closing.

Module F10-games-and-reductions

Security games, properly: the four parts and the advantage

F01 showed you a game box so the notation would stop being frightening. This card is the real thing: what a game **is**, what each part is doing, and the two advantage conventions that differ by a factor of two — which is the single most common arithmetic error in a first crypto proof.

Every game has exactly four parts

Whatever the security notion, the box on the page contains these and nothing else:

Part	What it does	Example (IND-CPA)
Setup	samples the secrets the adversary does not get	`K ←$ {0,1}^n`, `b ←$ {0,1}`
Oracles	the interface the adversary is allowed to poke	`Enc(m₀, m₁)`
The adversary's run	`A` executes, calling oracles	`b' ←$ A^{Enc}()`
The win condition	one predicate, decided by the game	`return (b' = b)`

Read a new definition by finding those four. **What changes between notions is almost never the shape — it is which oracles the adversary gets.** IND-CPA gives an encryption oracle; IND-CCA adds a decryption oracle; the unforgeability game gives a tagging oracle and forbids one particular output. Same skeleton.

> The definition is the theorem statement. If you cannot say precisely what the adversary is given and what it must produce, you cannot prove anything about it — and the course grades exactly that.

What an oracle actually models

An oracle is not a hint and not a weakness someone forgot to remove. It is a **modelling decision about what the real world hands an attacker.**

Giving `A` an encryption oracle says: *in deployment, an adversary can get messages of its choosing encrypted* — it induces traffic, it sends you an email you forward encrypted, it is a user of your own service. A scheme that fails IND-CPA fails against that real capability.

Two consequences that matter for proofs:

1. **`A` sees only the answers, never the state.** It does not learn `K`. This is the property every reduction leans on — see `f8-reductions`.
2. **The query count is part of the theorem, not bookkeeping.** Bounds look like `q(q−1)/2^{n+1}`. A scheme is not "secure" or "insecure"; it is secure *for adversaries making at most this many queries with this much time.* Dropping the resource accounting turns a theorem into a slogan.

Two conventions for advantage, and the factor of 2

Guess-the-bit. The game samples a hidden `b`, `A` outputs `b'`, and

```
Adv(A)  =  2 · Pr[b' = b]  −  1
```

The `2·(…) − 1` is a rescaling, not a fudge. Guessing at random gives `Pr = 1/2` and so `Adv = 0`; always right gives `Adv = 1`. Without it, a useless adversary would score `0.5` and you could not read the number.

Two-world. No hidden bit. The adversary is placed in one of two worlds and outputs a guess of which:

```
Adv(A)  =  | Pr[A ⇒ 1 in World 1]  −  Pr[A ⇒ 1 in World 0] |
```

The PRF game is written this way: World 1 is the real `F(K, ·)`, World 0 a truly random function.

They agree. If the challenger picks the world by a fair coin `b` and `A` wins when it names the world correctly:

```
Pr[b' = b] = ½·Pr[A ⇒ 1 | World 1] + ½·Pr[A ⇒ 0 | World 0]
           = ½·Pr[A ⇒ 1 | W1] + ½·(1 − Pr[A ⇒ 1 | W0])
           = ½ + ½·( Pr[A ⇒ 1 | W1] − Pr[A ⇒ 1 | W0] )
```

so `2·Pr[b'=b] − 1` is exactly the two-world difference. Same number, two spellings.

Where the 2 bites. When a proof bounds probabilities of *winning* and then converts to an advantage at the end, every bound picks up a factor of 2:

```
Pr[win] ≤ ½ + ε        ⟹        Adv = 2·Pr[win] − 1 ≤ 2ε
```

A hybrid whose hops cost `ε₁ + ε₂` in win-probability costs `2ε₁ + 2ε₂` in advantage. Losing that factor is not fatal to a proof's idea, but it is a wrong theorem, and it is marked as one. Decide which currency you are working in, write it down, and convert once, at the end.

Reading the superscript

`A^{Enc}` is not decoration. The superscript is the **complete list of what `A` may call** — the interface, in the type-signature sense. Anything not listed is unavailable, including the key.

When you meet `B^{f}` in a reduction, read it the same way: `B` has an oracle for `f` and nothing else. That constraint is the whole reason the reduction says something.

What to carry forward

1. Find the **four parts**. New notions differ by their oracle list, not shape.
2. **Oracles model real capability.** Query counts belong in the theorem.
3. Two advantage conventions, related by `Adv = 2·Pr[win] − 1`. **Pick one, say which, convert once.**
4. The superscript is the adversary's whole interface — and never includes the key.

Reductions: the four obligations, and what a proof owes

A reduction is the argument form this course is actually about. Nearly every theorem you will write has this shape, and nearly every mark you lose will be one of the four obligations below, skipped.

It is a contrapositive

You want: *if `F` is a secure PRF, then scheme `SE` built from it is IND-CPA secure.*

Proving that directly means quantifying over all adversaries and all their strategies, which is hopeless. So prove the contrapositive:

If some adversary `A` breaks `SE`, then some adversary `B` breaks `F`.

Since `F` is assumed secure, no such `B` exists, so no such `A` does either. `f1-contrapositive-proof` is the same move you already practised; the only new thing is that the objects being quantified over are algorithms.

The direction is the part people get backwards. You build an adversary against the *assumption* — the thing you are allowed to believe is hard — out of an adversary against the *construction*. Breaking is what gets transferred, and it travels from the new object to the old one.

The four obligations

A reduction is complete when it has supplied all four. Nothing else is required, and none of the four may be skipped.

1. Construct `B`

Write it as pseudocode, not prose. `B` is an adversary in the assumption's game, and its code is the proof's real content. If your write-up has no code box, you have described a reduction rather than given one.

2. Simulate `A`'s game using `B`'s own oracles

This is the load-bearing step. `A` expects a particular game; `B` must produce that game while holding only what the assumption's game gave it — usually one oracle and no key.

Two things must then be argued, and the second is the one that gets forgotten:

- **The simulation is faithful.** When `B`'s oracle is the real object, `A`'s view is *identically distributed* to its view in the real game. Not similar — identical, or with a stated statistical gap you then pay for.
- **`B` can actually run it.** Every line of the simulation uses only what `B` holds. A simulation that needs the key is not a reduction; it is a wish.

3. Relate the advantages

Produce an inequality, with the resource parameters in it:

```
Adv^{ind-cpa}_{SE}(A)  ≤  2 · Adv^{prf}_F(B)  +  q(q−1)/2^n
```

Not "so `B` also succeeds" — a number. This is where the two advantage conventions from `f8-security-games` matter: decide which currency you are in and convert exactly once.

The extra `q(q−1)/2^n` is typical. It is the part of `A`'s success that the assumption does **not** explain — here, a nonce collision — and it must be bounded separately and added on.

4. Account for `B`'s resources

State `B`'s running time and query count in terms of `A`'s. Usually `t_B ≈ t_A + q · (cost of one simulated query)` and `q_B = q_A`.

This is not bookkeeping. "There exists an adversary breaking `F`" is vacuous if that adversary takes `2^n` steps — every PRF has one of those. **The reduction only means something if `B` is efficient whenever `A` is.**

Tight, and why anyone cares

A reduction is **tight** when `B`'s advantage is close to `A`'s and its running time is close to `A`'s. A factor of `q` lost in the reduction is a real loss of security: if the proof gives `Adv_A ≤ q · Adv_B` and you want 128-bit security against an adversary making `2^40` queries, you need the underlying primitive to provide 168 bits. **Loose reductions are paid for in key sizes.**

You are not expected to produce optimally tight reductions. You are expected to know what your reduction's loss is, and to say so.

The failure modes, in the order they appear

1. **Reducing the wrong way** — building an adversary against the construction out of an adversary against the assumption. That proves nothing.
2. **A simulation that uses something `B` does not have** — most often the key.
3. **"Therefore `B` also wins"** with no inequality, so no theorem.
4. **Silence on resources**, leaving an unbounded `B` that proves nothing.
5. **Faithfulness asserted rather than argued** — the distributions are claimed identical without saying why, which hides the one step that can be wrong.

What to carry forward

Every reduction: **construct `B` · simulate `A`'s game from `B`'s oracles · relate the advantages with a number · account for the resources.**

Read `f8-listing-reduction` next — the same four obligations, with a real reduction on screen and a question for each one.

Hybrid arguments and game hopping

One reduction handles one assumption. Most real schemes lean on several things at once — a PRF, plus the fact that random nonces rarely repeat, plus a padding argument. A **hybrid argument** is how you use them one at a time.

The shape

Write a chain of games from the real world to an obviously-secure world:

```
G₀  =  the real security game
G₁  =  G₀, with one thing swapped
G₂  =  G₁, with the next thing swapped
   ⋮
Gₙ  =  a game where the adversary provably has advantage 0
```

Bound each hop separately, then add:

```
| Pr[G₀ ⇒ 1]  −  Pr[Gₙ ⇒ 1] |   ≤   Σᵢ | Pr[Gᵢ ⇒ 1] − Pr[Gᵢ₊₁ ⇒ 1] |
```

That is the triangle inequality you proved in `f2-prove-sd-triangle`, and it is the only reason the technique is sound. Without it there would be no argument that many small steps make a small total.

Change one thing per hop. Two changes in one hop means two justifications tangled together, and if the hop is wrong you cannot tell which half broke. A hop you cannot justify in one sentence is two hops.

Exactly three legal reasons for a hop

Every hop in the course is one of these. If yours is none of them, it is not justified yet.

1. Indistinguishability — a reduction

`Gᵢ` and `Gᵢ₊₁` differ by replacing a real object with an ideal one (a PRF with a random function, say). Justify with a reduction: an adversary distinguishing the two games becomes an adversary against the assumption.

Cost: `Adv^{assumption}(B)`.

2. Identical-until-bad — the difference lemma

`Gᵢ` and `Gᵢ₊₁` are the *same code* except on the branch where a flag `bad` is set. Then

```
| Pr[Gᵢ ⇒ 1] − Pr[Gᵢ₊₁ ⇒ 1] |  ≤  Pr[bad]
```

which is `f2-prove-difference-lemma`. The work moves to bounding `Pr[bad]`, and `bad` is usually a collision, so the bound is usually a birthday bound.

Cost: `Pr[bad]`.

3. A bridging step — no cost at all

The two games are *identically distributed*; the rewrite is for readability (inlining an oracle, sampling a value earlier, renaming). Say so explicitly and say why the distributions agree.

Cost: `0`. A bridging step you do not flag looks like an unjustified hop.

The endpoint has to be free

The chain is only worth building if the last game is one where the advantage is `0` *for reasons you can state* — usually because the adversary's entire view is independent of the hidden bit.

The one-time-pad move is the standard ending: once every ciphertext is a fresh uniform string, nothing in `A`'s view depends on `b`, so `Pr[b' = b] = 1/2` exactly. Not negligibly close to `1/2` — equal to it.

If you cannot argue the endpoint is free, you have not finished; you have relocated the problem.

Bookkeeping, which is where marks are lost

Three things to get right when you sum:

1. **One currency.** Win-probabilities or advantages, not both. Converting at the end costs a factor of 2 (`f8-security-games`); converting in the middle costs it more than once.

2. **Resources accumulate.** Each reduction has its own `B` with its own running time. State the total.
3. **Polynomially many hops.** If the number of hops depends on `q`, the sum is `q × (per-hop bound)` — the `poly × negligible` lemma from F06 says that is still negligible, but only if the hop count really is polynomial.

The shortest real chain there is

Take a one-time scheme built from a pseudorandom generator `G`, where the key is `G`'s seed and `Enc(K, m) = G(K) ⊕ m`. A single message, so no oracle:

	Game	Hop justified by	Cost
`G₀`	real scheme: `c = G(K) ⊕ m_b`	—	—
`G₁`	`G(K)` replaced by a uniform string `u`	reason 1 — reduction to PRG security	`Adv^{prg}(B)`
	`c = u ⊕ m_b`, so `Pr[win] = 1/2`	endpoint — one-time pad, view independent of `b`	`0`

Two lines and the theorem is done: `Adv(A) ≤ 2·Adv^{prg}(B)`.

Notice what each part contributed. **The reduction moved the assumption out of the way**; once `G(K)` became a uniform string there was no cryptography left, only the pad argument. That is what every chain is for — hop until the remaining claim is unconditional.

Longer chains are the same thing more times, and the reason they get longer is usually oracles: `q` queries mean `q` chances for something to repeat, and that brings in reason 2. `f8-table-gamehop` is that case, and `f8-prove-switching-lemma` proves the hardest hop of its kind.

What to carry forward

1. One change per hop, each with **one of three** justifications.
2. The endpoint must have advantage **exactly 0**, argued.
3. Sum in one currency, carry the resources, count the hops.

Module F11-python-for-crypto

Python for cryptography

CS 6260's projects are implementation work, and the language is Python. The mathematics is the hard part; the mechanics below are not hard, but getting them wrong costs hours and — in two specific cases — silently produces something insecure. This card is the short list.

`pow(a, e, n)` — never `a ** e % n`

Python's three-argument `pow` **is** repeated squaring, implemented in C.

```
pow(7, 128, 13)      # 3     — fast, bounded memory
(7 ** 128) % 13      # 3     — builds a 109-digit integer first
```

Both give 3 here. At RSA sizes the second does not finish, and it is not close: `m ** d` for a 2048-bit `d` is an integer with roughly `10^600` digits. There is not enough matter in the observable universe to store it.

Always the three-argument form. Since 3.8 it also inverts:

```
pow(a, -1, n)        # modular inverse of a mod n; raises if gcd(a, n) != 1
```

That raise is a feature — it is the solvability condition from F07 enforced by the runtime.

Integers are arbitrary precision

There is no overflow and no `int64`. A 4096-bit integer is an ordinary `int`. This is why Python is pleasant for this material: the numbers in your homework are just numbers.

```
n = 2**4096 - 1
n.bit_length()       # 4096
```

`bit_length()` is the input size from F05 — the thing "polynomial time" is polynomial *in*.

`bytes` vs `str`: keep them apart

Cryptography operates on **bytes**. Text is a bytes-plus-an-encoding.

```
b = "hello".encode("utf-8")   # str  -> bytes
s = b.decode("utf-8")         # bytes -> str
```

Hash and cipher APIs take `bytes` and refuse `str`. Take the `TypeError` as useful: it is asking which encoding you meant.

Indexing catches people out:

```
b = b"AB"
b[0]        # 65    -- an int
b[0:1]      # b'A'  -- a bytes of length 1
```

Iterating over `bytes` yields ints; slicing yields `bytes`.

Integers to bytes and back

```
n.to_bytes(length, "big")          # int -> bytes, fixed width
int.from_bytes(data, "big")        # bytes -> int
```

Fix `length` explicitly and use `"big"` consistently. Byte-order bugs produce plausible-looking wrong answers, which are the worst kind.

Randomness: `os.urandom` / `secrets`, never `random`

```
import os, secrets
key   = os.urandom(32)             # 32 cryptographically secure bytes
nonce = secrets.token_bytes(12)
k     = secrets.randbelow(n)       # uniform in [0, n)
```

`random` is a Mersenne Twister. It is not seeded securely and, worse, it is *invertible*: observe 624 consecutive outputs and you can reconstruct the internal state and predict every future value. It is excellent for simulations and disqualifying for keys.

Note the tell — this is a real instance of an F01 idea. `random` is perfectly uniform and passes statistical tests. Uniformity is not unpredictability. The security definition asks about an *adversary*, not a histogram.

Hashing

```
import hashlib, hmac
hashlib.sha256(b"data").hexdigest()
hmac.new(key, msg, hashlib.sha256).digest()
hmac.compare_digest(tag_a, tag_b)     # constant time
```

`compare_digest` matters. `==` on bytes short-circuits at the first differing byte, so its **running time leaks how many leading bytes matched** — enough to forge a tag one byte at a time. That is a side channel: a break that never touches the mathematics.

XOR on bytes

```
def xor(a: bytes, b: bytes) -> bytes:
    return bytes(x ^ y for x, y in zip(a, b))
```

`zip` stops at the shorter input. For a one-time pad that silently truncates, which is exactly the failure the OTP argument forbids — use `zip(a, b, strict=True)` (3.10+) and let it raise.

The four that actually bite

Mistake	Consequence
`a ** e % n`	never terminates at real sizes
`random` for keys	predictable keys; scheme is broken
`==` for tag comparison	timing side channel; forgeable
mixing `str` and `bytes`	`TypeError`, or worse, an encoding assumption

The first is a performance bug you will notice. The other three produce code that runs, passes your tests, and is insecure. Knowing them is part of knowing the subject, not separate from it.