Number Theory & Cryptographic Math

A Complete 10-Chapter Maths for Programmers Course

Topics covered:

Divisibility & modular arithmetic · the Euclidean & extended Euclidean algorithms

Primality testing · fast modular exponentiation

Euler's totient function & Fermat's Little Theorem · RSA

Capstone: building and breaking a working toy RSA implementation

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Number Theory Matters for Programmers

Number Theory & Cryptographic Math

Chapter 1 · Why Number Theory Matters for Programmers

Security's own `crypto1` (Cryptography Fundamentals) and `https1` (HTTPS/TLS Fundamentals) both use RSA and public-key cryptography constantly — and both, quite deliberately, never derive the actual mathematics underneath it. This course is that derivation: the specific slice of number theory that makes public-key cryptography work at all, built directly on Algorithms & Complexity's own growth-rate machinery and Discrete Mathematics Fundamentals' own proof techniques.

The Core Idea This Whole Course Builds Toward

Multiplying two numbers together is fast — genuinely, dramatically fast — no matter how large they are. Taking a large number and finding its two prime factors, with no other information to go on, is not. This asymmetry — **easy to multiply, hard to factor** — is not a minor curiosity. It is, quite literally, the single mathematical fact that public-key cryptography is built on top of.

VERIFIED DIRECTLY, AT A SMALL (ILLUSTRATIVE) SCALE

Two primes, `887` and `997`: verifying `887` is prime by trial division takes **28** divisions (up to `√887 ≈ 29`); verifying `997` takes **30** (up to `√997 ≈ 31`). Their product is `887 × 997 = 884,339` — computed in a single, trivial operation. But factoring `884,339` from scratch, with no prior knowledge of its factors, needs a trial-division search up to `√884339 ≈ 940` — roughly **32× more work** than checking either original prime alone, even at this tiny, three-digit scale.

WHY THIS SMALL DEMO ISN'T "PROOF" — AND WHY THAT'S HONEST

At three digits, a computer factors `884,339` instantly regardless of the ratio above — this demo illustrates the *mechanism*, not real-world hardness. The asymmetry becomes the actual foundation of security only once the primes involved are enormous — real RSA uses primes hundreds of digits long, where the gap between "multiply the two factors" and "factor the product with no shortcuts" stops being a curiosity and becomes computationally infeasible with any known classical algorithm. Chapter 6 builds the actual tool (primality testing) real systems use to find such primes without ever needing to factor anything.

Five Concrete Connections to Real Code

NUMBER THEORY TOPIC	WHERE IT ACTUALLY SHOWS UP
Modular arithmetic (Ch.3)	Hash table bucket indexing (`hash(key) % table_size`), circular buffers, clock/wraparound arithmetic
Modular arithmetic (Ch.3)	Checksums and error-detecting codes — the Luhn algorithm behind credit card validation, ISBN-10 and IBAN check digits
Modular exponentiation (Ch.7)	Pseudorandom number generation — classic Linear Congruential Generators run entirely on modular arithmetic
Primality testing (Ch.6)	Generating the large primes every real-world RSA key pair is built from
Everything combined (Ch.8–10)	Public-key cryptography itself — RSA key generation, encryption, and decryption, the central application this course builds toward

What This Course Won't Cover

Number theory as a full mathematical field is vast, and cryptography built on top of it is its own enormous discipline. This course deliberately covers only the specific machinery RSA needs, not a comprehensive tour of either:

- **A full academic number theory curriculum** — algebraic number theory, analytic number theory, and the deep theory of prime distribution stay out of scope
- **Elliptic-curve cryptography** — a genuinely different mathematical foundation (elliptic curve groups, not modular arithmetic on integers), substantial enough to deserve its own future treatment
- **The discrete logarithm problem in depth** — Diffie-Hellman key exchange leans on modular exponentiation (Chapter 7's own topic) but its security rests on a genuinely different hard problem than RSA's factoring problem; that distinction is named honestly where it comes up, not derived in full
- **Post-quantum cryptography** — an active, rapidly evolving research area built on entirely different mathematical foundations, out of scope for this course

> **WHY DRAW THE LINE AT RSA SPECIFICALLY**
>
> RSA is the single cryptographic system whose entire security argument can be built, honestly and completely, from divisibility, modular arithmetic, and a handful of classical theorems — exactly what this course has room to cover in ten chapters. Everything named above is either a genuinely different mathematical foundation (elliptic curves) or would require substantially more depth than a single course allows.

Where This Course Is Headed

CHAPTER	TOPIC
2	Divisibility & the Division Algorithm
3	Modular Arithmetic
4	The Euclidean Algorithm & GCD
5	The Extended Euclidean Algorithm & Modular Inverses
6	Prime Numbers & Primality Testing
7	Modular Exponentiation & Fast Exponentiation
8	Euler's Totient Function & Fermat's Little Theorem
9	RSA: How the Math Actually Works
10	Capstone — Building a Toy RSA Implementation

THIS COURSE'S THROUGHLINE

Every chapter adds exactly one piece a working RSA implementation genuinely needs — divisibility and remainders, working consistently under a modulus, finding a GCD efficiently, inverting under a modulus, telling whether a number is prime, exponentiating quickly under a modulus, and the two theorems that guarantee encryption and decryption actually undo each other. By Chapter 9, nothing about RSA is "magic" — it's eight prior chapters, assembled.

Hands-On Exercises

EXERCISE 1

Two primes, `101` and `103`, are multiplied to give `10,403`. Compute how many trial divisions are needed to verify `101` is prime (up to `√101`), and how many would be needed to factor `10,403` from scratch (up to `√10403`) with no prior knowledge of its factors. State the ratio between the two.

View solution

EXERCISE 2

A colleague says "modular arithmetic is just an academic curiosity — I've never needed it in real code." Using this chapter's own five connections, name two genuinely different real systems (not variations of the same idea) that quietly rely on modular arithmetic, and explain the connection for each.

View solution

EXERCISE 3

Using this chapter's own easy-to-multiply/hard-to-factor idea, explain in your own words why doubling the number of digits in an RSA key makes it dramatically harder to break, even though it only makes encryption and decryption modestly slower. (A precise answer isn't expected yet — Chapter 7's own fast-exponentiation material will make the "modestly slower" half exact; this exercise is about the intuition.)

View solution

CHAPTER 1 QUICK REFERENCE

- Core idea: multiplying is fast, factoring is hard — this asymmetry is the actual foundation of RSA
- Verified at small scale: factoring a product from scratch needs roughly `√(product)` work, dramatically more than verifying either factor alone once numbers are large
- Five direct connections: hash indexing, checksums/error-detecting codes, pseudorandom generation, primality testing for key generation, and RSA itself
- Deliberately out of scope: a full number theory curriculum, elliptic-curve cryptography, the discrete logarithm problem in depth, post-quantum cryptography
- Provides the mathematical foundation underneath Security's own `crypto1` and `https1` courses
- **Next chapter:** Divisibility and the division algorithm

CHAPTER 2 OF 10

Divisibility & the Division Algorithm

Number Theory & Cryptographic Math

Chapter 2 · Divisibility & the Division Algorithm

Everything in this course — modular arithmetic, the Euclidean algorithm, RSA itself — is ultimately built from one small idea: what does it actually mean for one whole number to divide another, and what can be guaranteed about the leftover when it doesn't divide evenly?

Divisibility, Defined Precisely

An integer `a` **divides** an integer `b` — written `a | b` — if there exists some integer `k` such that `b = a × k`. Read `a | b` as "`a` divides `b`," never as a fraction. `3 | 12` because `12 = 3 × 4`; `5 ∤ 12` (5 does *not* divide 12) because no integer `k` makes `5k = 12`.

The Division Algorithm: A Guarantee, Not Just an Operation

For any integer `a` (the dividend) and any positive integer `d` (the divisor), there exist **unique** integers `q` (quotient) and `r` (remainder) such that:

THE DIVISION ALGORITHM

`a = d × q + r`, where `0 ≤ r < d`. Both existence *and* uniqueness are guaranteed — there is exactly one `(q, r)` pair satisfying this for any given `a` and `d`, never more than one and never none.

The uniqueness is what makes this a genuine theorem rather than just a description of long division — `r` is pinned down precisely by the requirement `0 ≤ r < d`, ruling out every other way of splitting `a` into a multiple of `d` plus a leftover.

VERIFIED DIRECTLY, INCLUDING THE NEGATIVE-DIVIDEND CASE

`17 = 5×3 + 2` → `q=3, r=2`. `20 = 6×3 + 2` → `q=3, r=2`. And for a **negative** dividend: `-7 = 3×(-3) + 2` → `q=-3, r=2` — not `q=-2, r=-1`, since the rule `0 ≤ r < d` rules that pairing out even though `3×(-2)+(-1)=-7` is also arithmetically true. `-20 = 6×(-4) + 4` → `q=-4, r=4`.

A QUIET WIN FOR PYTHON, AND A PREVIEW OF CHAPTER 3

Python's own `//` and `%` already implement exactly this convention: `-7 // 3` gives `-3` and `-7 % 3` gives `2`, matching the mathematical division algorithm precisely. Not every language agrees — several truncate toward zero instead, giving a *negative* remainder for the same inputs. That difference matters enormously once remainders are used for anything beyond a leftover value, and Chapter 3 covers it directly.

Classic Divisibility Rules

Quick tests for divisibility by small numbers, without doing a full division — genuinely useful shortcuts, and a nice warm-up before the heavier machinery in later chapters:

DIVISOR	RULE
2	Last digit is even
3	Digit sum is divisible by 3
5	Last digit is 0 or 5
9	Digit sum is divisible by 9
10	Last digit is 0
11	Alternating digit sum (from the right) is divisible by 11

VERIFIED DIRECTLY, ON N = 1,458

Last digit 8 (even) → **divisible by 2**. Digit sum `1+4+5+8=18`, itself divisible by both 3 and 9 → **divisible by 3 and 9**. Last digit not 0 or 5 → **not divisible by 5**. Last digit not 0 → **not divisible by 10**. Alternating sum from the right (`8−5+4−1=6`) not divisible by 11 → **not divisible by 11**. Every rule matches the actual remainder computed directly.

Properties of Divisibility (Needed Later)

Three properties, each provable directly from the definition above (exactly the direct-proof style Discrete Mathematics Fundamentals built), that later chapters lean on without re-deriving:

- If `a | b` and `a | c`, then `a | (b + c)` and `a | (b − c)`
- If `a | b`, then `a | (bc)` for any integer `c`
- If `a | b` and `b | c`, then `a | c` (transitivity)

VERIFIED DIRECTLY

With `a=4`, `b=12`, `c=20`: `4|12` and `4|20`, and indeed `4|32` (their sum) and `4|(−8)` (their difference). Transitivity, with `3|9` and `9|27`: indeed `3|27`.

WHY THESE MATTER THIS EARLY

Chapter 4's Euclidean algorithm proof and Chapter 8's Fermat's Little Theorem proof both lean directly on the transitivity property above — it's introduced here, plainly, so it's already familiar when it does real work later.

Why This Is the Foundation Everything Else Builds On

Chapter 3's entire subject — modular arithmetic — is literally "take the division algorithm's own remainder, and build an entire arithmetic system that only ever tracks that remainder." Chapter 4's Euclidean algorithm is nothing more than the division algorithm, applied over and over. Every `%` operator in every programming language is a direct, real-world implementation of the guarantee stated in this chapter.

The Division Algorithm in Code

```
def division_algorithm(a, d):
    # Python's own // and % already satisfy 0 <= r < d for d > 0
    q = a // d
    r = a - d * q
    assert 0 <= r < d
    return q, r

print(division_algorithm(17, 5))    # (3, 2)
print(division_algorithm(-7, 3))    # (-3, 2) -- NOT (-2, -1)
print(division_algorithm(-20, 6))   # (-4, 4)
```

Hands-On Exercises

EXERCISE 1

Using the division algorithm's own convention (`0 ≤ r < d`), find the unique `q` and `r` for `a = −29` , `d = 4` . Show that `a = d×q + r` holds, and explain why `q=−7, r=−1` is not a valid answer even though `4×(−7)+(−1)=−29` is arithmetically true.

View solution

EXERCISE 2

Using this chapter's own divisibility rules, determine whether `n = 3,168` is divisible by 2, 3, 5, 9, 10, and 11 — show the specific rule check for each, then confirm every result against the actual remainder.

View solution

EXERCISE 3

Using this chapter's own divisibility properties, prove that if `7 | n`, then `7 | 5n` — and then, given that `7 | 21` and `7 | 5n` for some particular `n` where `5n = 105`, use the sum property to show `7 | (21 + 5n)` without dividing `126` directly.

View solution

CHAPTER 2 QUICK REFERENCE

- **Divisibility:** `a | b` means `b = a×k` for some integer `k`
- **Division algorithm:** for any integer `a` and positive `d`, unique `q, r` exist with `a = dq + r`, `0 ≤ r < d`
- Python's `//` / `%` already match this convention for negative dividends — not every language does (Chapter 3)
- Divisibility rules for 2/3/5/9/10/11 — quick shortcuts, all verified against real remainders
- Key properties: sum/difference rule, multiple rule, transitivity — used directly in Chapters 4 and 8
- **Next chapter:** Modular arithmetic

CHAPTER 3 OF 10

Modular Arithmetic

Number Theory & Cryptographic Math

Chapter 3 · Modular Arithmetic

Chapter 2 built the division algorithm and treated the remainder `r` as a byproduct — something left over after the "real" answer, the quotient. Modular arithmetic flips that entirely: the remainder *is* the whole answer, and the quotient is thrown away. Everything from here to RSA itself is built on that single reframing.

Congruence: "Same Remainder" as an Equals Sign

Two integers `a` and `b` are **congruent modulo** `n` — written `a ≡ b (mod n)` — if they leave the same remainder when divided by `n`, which is exactly equivalent to saying `n | (a − b)`.

VERIFIED DIRECTLY

`42 mod 13 = 3` and `3 mod 13 = 3` — same remainder, so `42 ≡ 3 (mod 13)`. Confirming the alternate definition: `13 | (42 − 3)`, since `42 − 3 = 39 = 13 × 3`.

Addition, Subtraction, and Multiplication All "Just Work"

The single most useful practical fact in this entire chapter: you can reduce modulo `n` at **any point** during a computation — before, during, or after — without ever changing the final answer modulo `n`.

THE REDUCE-ANYTIME PROPERTY

`(a + b) mod n = ((a mod n) + (b mod n)) mod n`, and the identical shape holds for subtraction and multiplication. Numbers can be kept small throughout a calculation instead of letting them grow to their full, unreduced size.

VERIFIED DIRECTLY, WITH LARGE NUMBERS

`a=987,654,321`, `b=123,456,789`, modulus `1,000`. Multiplying fully first, then reducing: `(a×b) mod 1000 = 269`. Reducing each operand first, then multiplying, then reducing again: `((a mod 1000)×(b mod 1000)) mod 1000 = 269` — **identical**. The same match holds for addition (`110` both ways) and subtraction (`532` both ways).

This is not a minor convenience — it's the entire reason modular exponentiation (Chapter 7) is computationally feasible at all for the enormous numbers RSA actually uses: intermediate values never need to

grow beyond the modulus itself, no matter how large the final unreduced answer would otherwise be.

Division Doesn't Work the Same Simple Way

Addition, subtraction, and multiplication all reduce cleanly under a modulus. Division does not have an equally simple counterpart — there's no general rule like `(a ÷ b) mod n = ((a mod n) ÷ (b mod n)) mod n`, because ordinary division isn't guaranteed to produce a whole number, and modular arithmetic only deals in integers.

A GENUINE GAP, DELIBERATELY LEFT OPEN HERE

"Dividing" under a modulus needs a completely different tool — a **modular inverse**, a number that behaves like `1/b` would, but only exists under specific conditions. Chapter 5 builds this directly, and it's one of the two pieces (alongside Chapter 7's fast exponentiation) that make RSA's own decryption step work at all.

A Real Cross-Language Gotcha: What `%` Actually Does With Negative Numbers

Chapter 2 already flagged this once — now it matters directly. Programming languages genuinely disagree about what `%` returns for a negative operand:

LANGUAGE	`-7 % 3`	CONVENTION
Python	`2`	Floored — result always shares the divisor's sign (matches Chapter 2's division algorithm exactly)
JavaScript	`-1`	Truncated toward zero — result shares the dividend's sign
Java	`-1`	Truncated toward zero
C / C++ (C99 and later)	`-1`	Truncated toward zero

VERIFIED DIRECTLY, SIMULATING BOTH CONVENTIONS

Python's native `-7 % 3` gives `2`. A simulated truncating-toward-zero remainder (the C/JavaScript/Java style) gives `-1` for the exact same inputs — the two conventions genuinely disagree, not just in presentation but in the actual returned value.

THE REAL BUG THIS CAUSES: A NEGATIVE ARRAY INDEX

A hash table with 10 buckets, using a hash that happens to produce `-23`: a truncating language's naive `hash % table_size` gives `-3` — a genuinely invalid, negative array index, and a real out-of-bounds risk in a language that allows negative indexing to silently wrap or crash. The fix, verified directly, works regardless of which convention the language uses: `((a % n) + n) % n` normalizes `-23` down to `7`, a valid bucket index every time.

Real Relevance: Clocks, Weekdays, and Wraparound

A 12-hour clock is arithmetic modulo 12; days of the week cycle modulo 7; a circular buffer's write position wraps modulo its own capacity. Any time a quantity needs to "wrap around" back to the start after reaching a fixed limit, that's modular arithmetic, whether or not the code ever calls it that.

Modular Arithmetic in Code

```
def mod_add(a, b, n): return (a + b) % n
def mod_sub(a, b, n): return (a - b) % n
def mod_mul(a, b, n): return (a * b) % n

# works even after reducing each operand first -- the reduce-anytime property
print(mod_mul(987654321, 123456789, 1000))                    # 269
print(mod_mul(987654321 % 1000, 123456789 % 1000, 1000))   # 269, same answer

# language-independent normalization -- always returns a value in [0, n)
def normalize_mod(a, n):
    return ((a % n) + n) % n

print(normalize_mod(-23, 10))   # 7 -- a valid, non-negative bucket index
```

Hands-On Exercises

EXERCISE 1

Determine whether `158 ≡ 38 (mod 12)`, using both the "same remainder" definition and the `n | (a−b)` definition. Show both checks agree.

View solution

EXERCISE 2

Using this chapter's own reduce-anytime property, compute `(456789 × 987654) mod 100` two ways: fully multiplying then reducing, and reducing each operand to its last two digits first, then multiplying, then reducing. Show both give the same result.

View solution

EXERCISE 3

A hash produces the value `-41` for some key, and the hash table has 8 buckets. Using this chapter's own normalization formula, compute the correct, valid bucket index — and explain, using this chapter's own truncating-vs-

floored comparison, what a language using truncation toward zero would compute for `-41 % 8` without normalization, and why that value is unusable as an index.

View solution

CHAPTER 3 QUICK REFERENCE

- **Congruence:** `a ≡ b (mod n)` means `a` and `b` share the same remainder mod `n`, equivalently `n | (a-b)`
- **Reduce-anytime property:** addition, subtraction, and multiplication can all be reduced mod `n` at any point without changing the final answer
- Division has no equally simple rule — needs a modular inverse (Chapter 5)
- **Cross-language gotcha:** Python floors (result matches divisor's sign); JavaScript/Java/C truncate toward zero (result matches dividend's sign) — genuinely different values for the same negative input
- **Universal fix:** `((a % n) + n) % n` always returns a value in `[0, n)`, regardless of language convention
- **Next chapter:** The Euclidean algorithm and GCD

CHAPTER 4 OF 10

The Euclidean Algorithm & GCD

Number Theory & Cryptographic Math

Chapter 4 · The Euclidean Algorithm & GCD

Chapter 3 closed with a gap: division doesn't reduce simply under a modulus, and closing that gap needs a modular inverse. Building a modular inverse needs the **extended** Euclidean algorithm (Chapter 5) — and that, in turn, is built directly on top of the plain **Euclidean algorithm** this chapter covers: the single fastest known way to find the greatest common divisor of two numbers.

GCD, and Why the Obvious Approach Doesn't Scale

The **greatest common divisor** of `a` and `b` — `gcd(a, b)` — is the largest positive integer that divides both. The obvious approach: check every integer from `min(a,b)` down to `1`, stopping at the first one that divides both. That works, but it needs up to `min(a,b)` checks — for the hundreds-of-digit numbers RSA actually uses, that's not slow, it's *completely infeasible*, worse than any brute-force approach Algorithms & Complexity ever flagged as impractical.

The Key Identity: `gcd(a, b) = gcd(b, a mod b)`

This is the entire algorithm, and it's not a shortcut or an approximation — it's an exact equality, provable directly from Chapter 2's own divisibility properties, not asserted on faith.

THE PROOF, USING ONLY CHAPTER 2'S OWN TOOLS

Let `a = bq + r` (Chapter 2's own division algorithm), so `r = a mod b`. Suppose `d` divides both `a` and `b`. Then `d | bq` (Chapter 2's multiple property), so `d | (a − bq)` (Chapter 2's subtraction property) — which is exactly `d | r`. So **every** common divisor of `(a, b)` is also a common divisor of `(b, r)`. Running the same argument in reverse (since `a = bq + r` also means any common divisor of `b` and `r` divides `a`) shows the two pairs have **exactly the same set** of common divisors — so they share the same greatest one.

VERIFIED DIRECTLY

Common divisors of `(48, 18)`: `{1, 2, 3, 6}`. Common divisors of `(18, 48 mod 18 = 12)`: `{1, 2, 3, 6}` — **identical sets**, exactly as the proof above guarantees.

The Algorithm, Traced

Repeatedly replace `(a, b)` with `(b, a mod b)` until `b` reaches `0` — at that point, `a` is the GCD (the base case: `gcd(a, 0) = a`).

TRACED AND VERIFIED DIRECTLY — GCD(48, 18)

`48 = 18×2 + 12` → next pair `(18, 12)`
`18 = 12×1 + 6` → next pair `(12, 6)`
`12 = 6×2 + 0` → remainder `0`, stop.
GCD(48, 18) = 6.

Why It Terminates So Quickly

Each step replaces the larger number with a strictly smaller remainder — Algorithms & Complexity's own recursion-analysis machinery applies directly. It's a provable fact (not covered in depth here, but well established) that the remainder more than halves every two steps in the worst case, giving the algorithm `O(log(min(a,b)))` steps overall — logarithmic in the size of the input, not linear.

VERIFIED DIRECTLY, ON TWO ~19-DIGIT RANDOM NUMBERS

GCD of two random 19-digit numbers finished in just **36 steps**, while $\log_2(\min(a,b)) \approx 60.8$ — comfortably within the logarithmic bound, for numbers that would take up to 10^{19} checks under the naive approach.

A GENUINELY INTERESTING WORST CASE: CONSECUTIVE FIBONACCI NUMBERS

Consecutive Fibonacci numbers are the classic slow case — `gcd(144, 89)` takes **10** steps, more than `gcd(144, 55)`'s **9** steps for a similarly-sized non-Fibonacci pair. But this is still nowhere near slow: **consecutive integers** are actually one of the *fastest* cases — `gcd(1000, 999)` finishes in just **2** steps, since `1000 mod 999 = 1` collapses the problem almost immediately. Even the "worst case" stays logarithmic; there's no input that makes this algorithm slow in the way brute-force factoring (Chapter 1) can be.

Real Relevance

Beyond reducing fractions to lowest terms, this exact algorithm is the direct foundation Chapter 5's extended Euclidean algorithm builds on to compute modular inverses — and RSA key generation itself runs a GCD check (`gcd(e, ϕ(n)) = 1`, Chapter 8) to confirm a chosen encryption exponent is actually usable, every single time a key pair is generated.

The Euclidean Algorithm in Code

```python
def gcd_recursive(a, b):
    if b == 0:
        return a
    return gcd_recursive(b, a % b)

def gcd_iterative(a, b):
    while b != 0:
        a, b = b, a % b
    return a

print(gcd_recursive(48, 18))   # 6
print(gcd_iterative(144, 89))   # 1 -- consecutive Fibonacci numbers are always coprime
```

Hands-On Exercises

EXERCISE 1

Trace the Euclidean algorithm step by step to find `gcd(252, 105)`, showing each `a = b×q + r` line and the resulting GCD.

View solution

EXERCISE 2

Using this chapter's own proof technique (Chapter 2's multiple and subtraction properties), show directly that any common divisor of `84` and `36` must also be a common divisor of `36` and `84 mod 36` — without simply listing out all the divisors of each number.

View solution

EXERCISE 3

Trace the Euclidean algorithm for the consecutive Fibonacci pair `gcd(55, 34)`, counting the number of steps, and compare it to `gcd(55, 21)` (not a consecutive Fibonacci pair, but a similarly-sized input). Which takes more steps, and does this match this chapter's own claim about Fibonacci numbers being a near-worst-case input?

View solution

CHAPTER 4 QUICK REFERENCE

- **Euclidean algorithm:** repeatedly replace `(a,b)` with `(b, a mod b)` until `b=0`; the last non-zero `a` is the GCD
- Provably correct: `gcd(a,b) = gcd(b, a mod b)`, proved directly from Chapter 2's own multiple and subtraction properties

- **O(log(min(a,b)))** steps — dramatically faster than brute-force checking down from `min(a,b)`
- Consecutive Fibonacci numbers are a genuine near-worst case; consecutive integers are one of the fastest cases
- Direct foundation for Chapter 5's extended Euclidean algorithm and for RSA's own key-generation GCD check
- **Next chapter:** The extended Euclidean algorithm and modular inverses

CHAPTER 5 OF 10

The Extended Euclidean Algorithm & Modular Inverses

Number Theory & Cryptographic Math

Chapter 5 · The Extended Euclidean Algorithm & Modular Inverses

Chapter 4 built the Euclidean algorithm to find *a number* — the GCD. This chapter extends it to find something far more useful: two numbers that *combine* to produce that GCD. That small extension is the entire tool needed to close the gap Chapter 3 left open — dividing under a modulus.

Bézout's Identity

THE THEOREM

For any integers `a` and `b`, there exist integers `x` and `y` such that `ax + by = gcd(a, b)`. These `x, y` are called **Bézout coefficients**.

This isn't just a useful trick — it's a genuine guarantee that a specific linear combination of `a` and `b` always exists and always lands exactly on their GCD, never anything smaller.

The Extended Euclidean Algorithm: Finding x and y

Run the same division steps as Chapter 4, but track two extra running values (`s` and `t`) at every step, each starting from the trivial coefficients of `a` and `b` themselves, updated by the same quotient used to update the remainder.

TRACED AND VERIFIED DIRECTLY — A GENERAL (NON-COPRIME) EXAMPLE, GCD(240, 46)

`q=5: r=46, s=0, t=1`
`q=4: r=10, s=1, t=-5`
`q=1: r=6, s=-4, t=21`
`q=1: r=4, s=5, t=-26`
`q=2: r=2, s=-9, t=47`

Final: `gcd(240,46)=2`, with `x=-9, y=47`. Check: `240×(−9) + 46×47 = −2160 + 2162 = 2` — exactly the GCD, confirmed directly.

Modular Inverses: What Bézout Actually Unlocks

A number `a` has a **modular inverse** mod `n` — a number `a⁻¹` such that `a × a⁻¹ ≡ 1 (mod n)` — if and only if `gcd(a, n) = 1` (`a` and `n` are **coprime**).

WHY THIS FOLLOWS DIRECTLY FROM BÉZOUT — NO NEW THEORY NEEDED

If `gcd(a, n) = 1`, Bézout guarantees integers `x, y` with `ax + ny = 1`. Reducing both sides mod `n`: the `ny` term vanishes (it's a multiple of `n`), leaving `ax ≡ 1 (mod n)` — `x`, reduced into `[0, n)`, *is* the modular inverse. The extended Euclidean algorithm doesn't just prove an inverse exists — it directly computes it.

TRACED AND VERIFIED DIRECTLY — THE MODULAR INVERSE OF 17 MOD 43

`gcd(43, 17)`: `q=2: r=17,s=0,t=1` · `q=1: r=9,s=1,t=-2` · `q=1: r=8,s=-1,t=3` · `q=8: r=1,s=2,t=-5`. Final: `gcd=1`, with `s=2` (coefficient of 43) and `t=-5` (coefficient of 17). Check: `43×2 + 17×(-5) = 86 − 85 = 1`. The modular inverse of `17 mod 43` is `t mod 43 = -5 mod 43 = 38`. Verified: `17 × 38 mod 43 = 646 mod 43 = 1`.

When No Inverse Exists

If `gcd(a, n) ≠ 1`, no modular inverse exists at all — not "hard to find," genuinely absent.

VERIFIED DIRECTLY — A COUNTEREXAMPLE, 4 MOD 8

`gcd(4, 8) = 4 ≠ 1`. Checking *every* possible `x` from 0 to 7: `4x mod 8` only ever produces `0` or `4` — **never 1**, for any of the 8 possible values of `x`. No inverse exists, exactly as the `gcd=1` condition predicts.

Real Relevance: This Is Literally RSA's Private Key

Chapter 9 computes RSA's private decryption exponent as `d = e⁻¹ mod ϕ(n)` — a modular inverse, found using exactly the algorithm in this chapter. Key generation deliberately *chooses* `e` to be coprime with `ϕ(n)` specifically so this inverse is guaranteed to exist — the `gcd = 1` condition from this chapter is a real, load-bearing constraint on how RSA keys are generated, not a footnote.

The Extended Euclidean Algorithm in Code

```python
def extended_gcd(a, b):
    old_r, r = a, b
    old_s, s = 1, 0
    old_t, t = 0, 1
    while r != 0:
        q = old_r // r
        old_r, r = r, old_r - q * r
        old_s, s = s, old_s - q * s
        old_t, t = t, old_t - q * t
    return old_r, old_s, old_t   # gcd, x, y -- with a*x + b*y == gcd

def mod_inverse(a, n):
    g, x, _ = extended_gcd(a, n)
    if g != 1:
        raise ValueError(f"no inverse: gcd({a},{n}) = {g}, not 1")
    return x % n

print(mod_inverse(17, 43))   # 38
print(mod_inverse(4, 8))     # raises ValueError -- gcd(4,8) = 4
```

Hands-On Exercises

EXERCISE 1

Using the extended Euclidean algorithm, find Bézout coefficients `x, y` for `gcd(99, 78)`. Show each step's `q, r, s, t`, and verify `99x + 78y` equals the GCD you found.

View solution

EXERCISE 2

Find the modular inverse of `7` mod `26` using the extended Euclidean algorithm, showing every step, and verify `7 × (inverse) mod 26 = 1`.

View solution

EXERCISE 3

Without running the extended Euclidean algorithm, determine whether `6` has a modular inverse mod `15`. Justify your answer using this chapter's own existence condition, then confirm it by checking every possible `x` from 0 to 14 for `6x mod 15`.

View solution

CHAPTER 5 QUICK REFERENCE

- **Bézout's identity:** for any `a, b`, integers `x, y` exist with `ax + by = gcd(a,b)`
- **Extended Euclidean algorithm:** track running coefficients `s, t` alongside the normal GCD steps
- **Modular inverse exists iff** `gcd(a,n)=1` — proved directly from Bézout, not a separate theorem
- When `gcd(a,n)≠1`, no inverse exists at all — verified directly by exhaustively checking every candidate
- This chapter's algorithm computes RSA's own private exponent: `d = e⁻¹ mod ϕ(n)` (Chapter 9)
- **Next chapter:** Prime numbers and primality testing

CHAPTER 6 OF 10

Prime Numbers & Primality Testing

Number Theory & Cryptographic Math

Chapter 6 · Prime Numbers & Primality Testing

Chapter 5's modular inverses need `gcd(a, n) = 1`. The cleanest way to guarantee that is to build `n` out of **prime** numbers — and RSA does exactly this. This chapter builds the actual tool real systems use to find primes large enough for cryptography.

What a Prime Actually Is

A **prime** is an integer greater than 1 whose only positive divisors are 1 and itself. `1` is deliberately excluded by definition — if it counted as prime, numbers wouldn't have a single unique prime factorization (`12 = 2×2×3` could also be written `1×2×2×3`, `1×1×2×2×3`, and so on, endlessly).

Trial Division, Formalized

Chapter 1's own finding-box already used trial division informally. Here's why checking only up to `√n` is enough, not just a shortcut:

THE PROOF

If `n = a × b` with `a ≤ b`, then `a` cannot be greater than `√n` — if it were, `b ≥ a > √n` too, making `a × b > n`, a contradiction. So if `n` has any factor at all, its *smaller* factor is guaranteed to be at most `√n`. Checking every candidate up to `√n` is therefore not a heuristic — it's a complete search.

The Sieve of Eratosthenes: Finding Many Primes at Once

Trial division checks *one* number. To find *every* prime up to some limit, the Sieve of Eratosthenes is dramatically more efficient: starting from 2, cross out every multiple of each prime found, moving upward.

VERIFIED DIRECTLY — SIEVING UP TO 50

`2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47` — **15** primes, each confirmed by direct trial division against the same list.

Trial division is the right tool for checking a single already-chosen candidate; the sieve is the right tool for finding many small primes at once — genuinely different jobs, not competing solutions to the same problem.

Why Neither Tool Works at RSA Scale

THE SCALING WALL

Real RSA primes are hundreds of digits long. Even `√n` for a 600-digit number is still roughly a 300-digit number — trial division up to that point, or sieving a range that large, is completely infeasible, not merely slow, exactly the same wall Chapter 1's own multiply-vs-factor asymmetry described from the attacker's side. Generating primes at cryptographic scale needs a fundamentally different approach.

Probabilistic Primality Testing: Trading Certainty for Speed

Instead of proving primality with total certainty, a **probabilistic** test checks a property that *every* prime satisfies, but which most composite numbers fail. A candidate that passes many independent rounds is declared "probably prime," with a failure probability that can be made astronomically small — smaller than the odds of an undetected hardware error corrupting the computation itself.

THE STARTING IDEA: FERMAT'S OWN TEST

Fermat's Little Theorem (proved fully in Chapter 8) states that for a genuine prime `p` and any `a` not divisible by `p`: `a^(p-1) ≡ 1 (mod p)`. The simplest probabilistic test just checks this directly for a random base `a` — if it fails, `n` is definitely composite; if it passes, `n` is *probably* prime. Miller-Rabin (the real algorithm production systems use) refines this same core idea with a stronger, harder-to-fool check — its full derivation is beyond this course's own scope, but the underlying trade-off is exactly what's shown here.

Why a Single Test Isn't Enough: A Real Counterexample

VERIFIED DIRECTLY — 341, THE SMALLEST FERMAT PSEUDOPRIME TO BASE 2

`341 = 11 × 31` — genuinely composite. Yet the plain Fermat test with base `a=2`: `2^340 mod 341 = 1` — it **passes**, wrongly suggesting 341 might be prime. Testing with other bases catches it immediately: base `3` gives `56`, base `5` gives `67`, base `7` gives `56` — all correctly fail. A single base can be fooled; this is exactly why real primality testing runs *many* independent random bases, driving the false-positive probability down exponentially with each additional round.

Real Relevance

Every real-world crypto library — the ones underneath Security's own `crypto1` and `https1` — generates RSA primes by sieving out small factors quickly, then running many rounds of a probabilistic test (Miller-Rabin in practice) on the survivors. Nothing here is a simplification for teaching purposes; this is genuinely how production key generation works.

Primality Testing in Code

```
import math

def is_prime_trial(n):
    if n < 2: return False
    for i in range(2, math.isqrt(n) + 1):
        if n % i == 0:
            return False
    return True

def sieve_of_eratosthenes(limit):
    is_prime = [True] * (limit + 1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, math.isqrt(limit) + 1):
        if is_prime[i]:
            for multiple in range(i*i, limit + 1, i):
                is_prime[multiple] = False
    return [i for i in range(2, limit + 1) if is_prime[i]]

def fermat_test(n, a):
    return pow(a, n - 1, n) == 1   # fast modular exponentiation -- Chapter 7

print(fermat_test(341, 2))   # True -- fooled! 341 is actually composite
print(fermat_test(341, 3))   # False -- correctly catches it
```

Hands-On Exercises

EXERCISE 1

Using this chapter's own `√n` proof, determine the largest candidate divisor that needs to be checked to verify whether `n = 221` is prime. Then use trial division up to that bound to determine whether 221 is prime, and if not, name its factors.

View solution

EXERCISE 2

Using the Sieve of Eratosthenes method from this chapter, find every prime up to 30, showing which multiples get crossed out by each prime as the sieve proceeds (starting with 2, then 3, then 5).

View solution

EXERCISE 3

Explain, using this chapter's own 341 counterexample, why a real-world crypto library would never trust a single Fermat test with a single fixed base (such as always testing with `a=2`) when generating RSA primes — and why running the test with several different random bases makes this specific failure far less likely to matter in practice.

View solution

CHAPTER 6 QUICK REFERENCE

- **Prime:** an integer >1 with only 1 and itself as divisors — 1 excluded to preserve unique factorization
- **Trial division:** only needs to check up to `√n` — proved directly, not just assumed
- **Sieve of Eratosthenes:** the right tool for finding many primes in a range at once, not for checking one large candidate
- Both tools are infeasible at RSA's real scale (hundreds-of-digit primes) — the same wall as Chapter 1's own multiply-vs-factor asymmetry
- **Probabilistic testing (Fermat/Miller-Rabin):** trades absolute certainty for speed — a false positive's probability shrinks with each additional random base tested
- 341 is a real, verified Fermat pseudoprime to base 2 — a single test can be fooled, which is exactly why multiple random bases are used in practice
- **Next chapter:** Modular exponentiation and fast exponentiation

CHAPTER 7 OF 10

Modular Exponentiation & Fast Exponentiation

Number Theory & Cryptographic Math

Chapter 7 · Modular Exponentiation & Fast Exponentiation

Chapter 6's code quietly used Python's built-in three-argument `pow(a, b, n)` without explaining how it computes $a^b \bmod n$ so fast even when `b` is astronomically large. This chapter opens that box — and the answer turns out to be a direct, concrete payoff of Algorithms & Complexity's own recursion and growth-rate machinery.

The Naive Approach, and Why It Can't Scale

The obvious way to compute $a^b \bmod n$: multiply by `a`, one step at a time, `b` times, reducing mod `n` along the way. That's `O(b)` **multiplications** — linear in the exponent itself, not the number of *digits* in the exponent.

> **WHY THIS GENUINELY CAN'T WORK AT RSA SCALE**
>
> A real 2048-bit RSA exponent can be a number with roughly 617 *decimal digits*. The naive approach needs a number of multiplications equal to that exponent's own *value* — not its digit count. That's not "slow," it's a number of operations vastly larger than the number of atoms in the observable universe.

Fast Exponentiation: Square-and-Multiply

The fix reuses the exact same halving idea behind Algorithms & Complexity's own `T(n) = T(n/2) + O(1)` recurrence: a^b can always be built from `a^(b/2)`, squared — halving the exponent at every step instead of decrementing it by one.

> **THE RECURSIVE IDEA**
>
> If `b` is even: $a^b = (a^{b/2})^2$. If `b` is odd: $a^b = a \times (a^{(b-1)/2})^2$. Either way, the exponent roughly halves every step — `O(log b)` **multiplications** total, exactly Algorithms & Complexity's own logarithmic-recursion shape.

In practice, this is implemented iteratively by walking through `b`'s own binary digits — squaring a running `base` value at every step, and multiplying it into the running `result` exactly when the current bit is `1`.

> **TRACED AND VERIFIED DIRECTLY — $3^{13} \bmod 7$ (13 = 1101 IN BINARY)**

`bit=1: result = 1×3 mod 7 = 3`, square base: `3² mod 7 = 2`
`bit=0: (no multiply)`, square base: `2² mod 7 = 4`
`bit=1: result = 3×4 mod 7 = 5`, square base: `4² mod 7 = 2`
`bit=1: result = 5×2 mod 7 = 3`, square base: `2² mod 7 = 4`
Final result: **3** — matching the naive method's own $3^{13} \bmod 7$ result exactly, computed in **7** total multiplications (4 squarings + 3 bit-multiplies) instead of the naive method's **13**.

The Gap, at Real Scale

EXPONENT B	NAIVE: O(B) MULTIPLICATIONS	FAST: O(LOG B) MULTIPLICATIONS
13	13	7
100	100	10
1,000	1,000	16
2^{2048} (real 2048-bit RSA scale, 617 digits)	~10^{617} — physically impossible	**2,050**

All values verified directly. At real RSA scale (`b = 2²⁰⁴⁸`, a number with a single bit set among its 2,049 binary digits), fast exponentiation needs one squaring per bit position plus one extra multiply for that single set bit — `2,049 + 1 = 2,050` total, matching the verified count exactly. A computation any modern computer finishes in milliseconds, for a naive approach that would never finish at all.

The Other Half of the Trick: Reduce After *Every* Multiplication

Fast exponentiation alone isn't enough — Chapter 3's own reduce-anytime property has to be applied at *every single step*, not just at the end, or the raw numbers explode before the exponent ever finishes shrinking.

VERIFIED DIRECTLY — THE DIGIT-LENGTH EXPLOSION, WITH AND WITHOUT REDUCING EVERY STEP

Repeatedly squaring `3` without ever reducing: `1, 2, 4, 8, 16, 31` digits after each successive squaring — **doubling** every step. After just 11 squarings, the raw unreduced number has **978 digits**. Reducing mod `n` after every single squaring, the value never exceeds `n`'s own digit count — **9 digits**, the entire way through, no matter how many squarings run.

Real Relevance: This Is Literally How RSA Encrypts and Decrypts

Chapter 9 defines RSA encryption as `c = mᵉ mod n` and decryption as `m = cᵈ mod n` — both are modular exponentiations, computed with the exact algorithm in this chapter, on numbers hundreds of digits long. Fast exponentiation isn't an optimization RSA happens to use — it's the specific reason RSA is fast enough to be usable at all.

Fast Modular Exponentiation in Code

```
def fast_modpow(a, b, n):
    result = 1
    base = a % n
    while b > 0:
        if b & 1:                       # current bit is 1
            result = (result * base) % n
        base = (base * base) % n     # reduce EVERY step -- Chapter 3
        b >>= 1                           # move to the next bit
    return result

print(fast_modpow(3, 13, 7))              # 3
print(fast_modpow(3, 13, 7) == pow(3, 13, 7))  # True -- matches Python's own built-in
```

Hands-On Exercises

EXERCISE 1

Using the square-and-multiply method, trace $5^9 \bmod 11$ step by step (write out 9 in binary first). Show each squaring and each bit-triggered multiply, and state the final result and total multiplication count.

View solution

EXERCISE 2

For an exponent $b = 10{,}000$, compute how many multiplications the naive approach needs, and roughly how many the fast approach needs (using $\log_2(b)$). Compute the ratio between them.

View solution

EXERCISE 3

Explain, using this chapter's own digit-length findings, why a fast exponentiation implementation that forgets to reduce mod n after every squaring (and only reduces once at the very end) would still be dramatically slower than the correct version — even though it uses the exact same $O(\log b)$ number of multiplications.

View solution

CHAPTER 7 QUICK REFERENCE

- **Naive exponentiation:** $O(b)$ multiplications — infeasible for RSA-scale exponents

- **Fast exponentiation (square-and-multiply):** `O(log b)` multiplications — walks the exponent's own binary digits
- Verified at 2048-bit RSA scale: **2,050** multiplications, vs. a physically impossible ~10^{617} for the naive approach
- **Reduce mod n after every single multiplication** — not just at the end — or the raw numbers double in digit-length every step
- This is the exact algorithm behind RSA's own `c = m^e mod n` encryption and `m = c^d mod n` decryption (Chapter 9)
- **Next chapter:** Euler's totient function and Fermat's Little Theorem

CHAPTER 8 OF 10

Euler's Totient Function & Fermat's Little Theorem

Number Theory & Cryptographic Math

Chapter 8 · Euler's Totient Function & Fermat's Little Theorem

Every prior chapter built a tool: the division algorithm, modular arithmetic, the Euclidean algorithm, modular inverses, primality testing, fast exponentiation. This chapter builds the last piece of *pure theorem* — the actual mathematical fact that makes RSA's encryption and decryption genuinely undo each other, not just a fact stated on faith.

Euler's Totient Function: Counting Coprimes

Euler's totient function, `ϕ(n)`, counts how many integers in `[1, n]` are coprime to `n` — that is, how many satisfy `gcd(k, n) = 1`.

VERIFIED DIRECTLY — Φ(P) FOR A PRIME

For a prime `p`, *every* integer from `1` to `p-1` is automatically coprime to `p` (a prime has no divisors besides 1 and itself), so `ϕ(p) = p − 1`. Confirmed directly: `ϕ(5)=4`, `ϕ(7)=6`, `ϕ(11)=10`, `ϕ(13)=12` — each exactly matching `p-1`.

THE FORMULA RSA ACTUALLY NEEDS: Φ(PQ) FOR TWO DISTINCT PRIMES

`ϕ` is **multiplicative** for coprime inputs: `ϕ(mn) = ϕ(m) × ϕ(n)` whenever `gcd(m, n) = 1`. Since two distinct primes are always coprime to each other, this gives `ϕ(p × q) = (p−1)(q−1)` directly.

VERIFIED DIRECTLY

`ϕ(3×5=15) = 8`, and `(3−1)(5−1) = 8` — match. `ϕ(5×7=35) = 24`, and `(5−1)(7−1) = 24` — match. `ϕ(11×13=143) = 120`, and `(11−1)(13−1) = 120` — match.

WHY "COPRIME" IS A REAL REQUIREMENT, NOT FINE PRINT

`ϕ(4)=2` and `ϕ(6)=2`, so the naive product would suggest `ϕ(24)=4`. The *actual* value is `ϕ(24)=8` — the formula genuinely fails once `gcd(m,n)≠1` (here `gcd(4,6)=2`). This is exactly why RSA's own security depends on using two **distinct** primes, not any two coprime-looking numbers.

Fermat's Little Theorem, Proved — Not Just Stated

THE THEOREM

For a prime `p` and any integer `a` not divisible by `p`: `a^(p-1) ≡ 1 (mod p)`.

Chapters 6 and 7 both used this without proof. Here's the actual argument, built entirely from tools this course has already established:

THE PROOF — A PERMUTATION ARGUMENT, VERIFIED DIRECTLY ON P=7, A=3

Since `gcd(a,p)=1`, multiplying every element of `{1, 2, ..., p-1}` by `a` (mod `p`) just **rearranges** the same set — it never produces a repeat or a zero. Verified: `{3,6,9,12,15,18} mod 7 = {3,6,2,5,1,4}` — exactly a reordering of `{1,2,3,4,5,6}`. Multiplying every element of both sets together must therefore give the same product mod `p`: `a^(p-1) × (p-1)! ≡ (p-1)! (mod p)` — verified directly, both sides equal `6`. Since `(p-1)!` is coprime to `p` (a product of numbers all coprime to `p`), Chapter 5's own modular inverse can **cancel it from both sides**, leaving exactly `a^(p-1) ≡ 1 (mod p)`.

This is a genuine direct proof, in the same style Discrete Mathematics Fundamentals established — no step taken on faith, and the final cancellation step is literally Chapter 5's own modular inverse doing real work.

VERIFIED DIRECTLY, EXHAUSTIVELY

Checked for every `a` from `1` to `p-1`, for `p ∈ {5, 7, 11, 13}`: `a^(p-1) mod p = 1` in **every single case**, no exceptions found.

Euler's Theorem: The Generalization to Any Modulus

THE THEOREM

For any `n` and any `a` with `gcd(a, n) = 1`: `a^ϕ(n) ≡ 1 (mod n)`.

The exact same permutation argument works, replacing "all of `1` to `p-1`" with "everything in `[1,n]` coprime to `n`" — that set has `ϕ(n)` elements by definition, and multiplying by any coprime `a` still just rearranges it. Fermat's Little Theorem is the special case where `n` happens to be prime (since `ϕ(p) = p-1`).

VERIFIED DIRECTLY — EULER'S THEOREM ON A COMPOSITE MODULUS, N=15

`ϕ(15) = 8`. Checking every `a` coprime to 15 (`1, 2, 4, 7, 8, 11, 13, 14`): `a⁸ mod 15 = 1` in **every single case**.

Real Relevance: This Is RSA's Actual Correctness Argument

Chapter 9 sets `n = pq`, uses `ϕ(n) = (p−1)(q−1)` (this chapter's own formula), and chooses `e` and `d` so that `ed ≡ 1 (mod ϕ(n))` — a modular inverse, Chapter 5's own tool. Euler's theorem then guarantees `m^(ed) ≡ m (mod n)` for any message `m` coprime to `n` — **the entire reason decrypting an RSA-encrypted message recovers the original message**. Nothing about that guarantee is new material — it's this chapter's own theorem, applied.

Totient & Fermat's Little Theorem in Code

```
import math

def totient(n):
    return sum(1 for k in range(1, n + 1) if math.gcd(k, n) == 1)

def totient_from_primes(p, q):
    return (p - 1) * (q - 1)   # requires p, q distinct primes

print(totient(15), totient_from_primes(3, 5))   # 8 8

# Fermat's Little Theorem, verified for every a in a prime's own range
p = 13
print(all(pow(a, p - 1, p) == 1 for a in range(1, p)))   # True

# Euler's theorem, verified on a composite modulus
n = 15
phi_n = totient(n)
print(all(pow(a, phi_n, n) == 1 for a in range(1, n) if math.gcd(a, n) == 1))   # True
```

Hands-On Exercises

EXERCISE 1

Compute `ϕ(17 × 19)` using this chapter's own formula, then verify it by directly counting the integers from 1 to `17×19` that are coprime to `17×19`.

View solution

EXERCISE 2

Using this chapter's own permutation argument, verify Fermat's Little Theorem for `p=11`, `a=4`: show that `{4, 8, 12, ..., 40} mod 11` is a permutation of `{1,...,10}`, then confirm `4^10 mod 11 = 1`.

View solution

EXERCISE 3

Explain why Fermat's Little Theorem is a special case of Euler's theorem, using this chapter's own φ(p)=p−1 fact, and verify Euler's theorem directly for the composite modulus `n=21` (compute φ(21) first, then check it for at least three different values of `a` coprime to 21).

View solution

CHAPTER 8 QUICK REFERENCE

- **Euler's totient φ(n):** the count of integers in [1,n] coprime to n
- `ϕ(p) = p−1` for prime p; `ϕ(pq) = (p−1)(q−1)` for distinct primes — the exact formula RSA needs
- The multiplicative property genuinely requires coprimality — verified to fail otherwise (φ(24) ≠ φ(4)×φ(6))
- **Fermat's Little Theorem:** `a^(p−1) ≡ 1 (mod p)` for prime p — proved directly via a permutation argument plus Chapter 5's own modular inverse
- **Euler's theorem:** `a^ϕ(n) ≡ 1 (mod n)` for any n with gcd(a,n)=1 — Fermat's theorem is the prime special case
- This is RSA's own core correctness guarantee: `m^(ed) ≡ m (mod n)`, assembled fully in Chapter 9
- **Next chapter:** RSA — how the math actually works

CHAPTER 9 OF 10

RSA: How the Math Actually Works

Number Theory & Cryptographic Math

Chapter 9 · RSA: How the Math Actually Works

No new mathematics appears in this chapter. Every tool RSA needs was already built — divisibility (Ch.2), modular arithmetic (Ch.3), the Euclidean algorithm (Ch.4), modular inverses (Ch.5), primality testing (Ch.6), fast exponentiation (Ch.7), and Euler's theorem (Ch.8). This chapter is pure assembly.

Key Generation, Step by Step

1 — CHOOSE TWO DISTINCT PRIMES (CH.6)

Pick `p` and `q` — real systems use primality testing on hundreds-of-digit candidates; this chapter's worked example uses `p=11`, `q=13` for full traceability.

2 — COMPUTE THE MODULUS N = PQ

`n = 11 × 13 = 143`. This becomes part of both the public and private key.

3 — COMPUTE Φ(N) = (P−1)(Q−1) (CH.8)

`ϕ(143) = 10 × 12 = 120`. This value stays secret — it's the entire reason factoring `n` would break the system (see the security section below).

4 — CHOOSE E WITH GCD(E, Φ(N)) = 1 (CH.4)

`e = 7`. Checked directly: `gcd(7, 120) = 1` — this is exactly Chapter 4's own GCD check, and it's what guarantees the next step's inverse will actually exist (Chapter 5's own existence condition).

5 — COMPUTE D = E^{-1} MOD Φ(N) (CH.5)

Running the extended Euclidean algorithm on `(7, 120)`: `d = 103`. Verified directly: `7 × 103 mod 120 = 721 mod 120 = 1`.

6 — PUBLISH AND KEEP SECRET

Public key: `(n, e) = (143, 7)` — shared with anyone. **Private key:** `(n, d) = (143, 103)` — kept secret. `p`, `q`, and `ϕ(n)` are also destroyed or kept secret — they're no longer needed once `d` is computed.

Encryption and Decryption

BOTH ARE JUST CHAPTER 7'S OWN FAST MODULAR EXPONENTIATION

Encrypt: `c = m^e mod n`, using the public key. **Decrypt:** `m = c^d mod n`, using the private key. Nothing else — the entire "encryption algorithm" is a single call to the exact function Chapter 7 built.

VERIFIED DIRECTLY — THE FULL ROUND TRIP, M=9

Encrypt: `c = 9⁷ mod 143 = 48`. Decrypt: `m = 48¹⁰³ mod 143 = 9` — the **original message, recovered exactly**. Confirmed for five more messages: `m=2→c=128→2`, `m=5→c=47→5`, `m=10→c=10→10`, `m=20→c=136→20`, `m=50→c=41→50` — every single one round-trips correctly.

Why It Works: The Actual Proof, Not an Assertion

By construction (step 5), `ed ≡ 1 (mod ϕ(n))` — meaning `ed = 1 + kϕ(n)` for some integer `k`. Decrypting an encrypted message:

THE FULL DERIVATION, USING ONLY CHAPTER 8'S OWN THEOREM

`c^d = (m^e)^d = m^(ed) = m^(1 + kϕ(n)) = m × (m^ϕ(n))^k`. By **Euler's theorem (Chapter 8)**, `m^ϕ(n) ≡ 1 (mod n)` whenever `gcd(m,n)=1` — so `(m^ϕ(n))^k ≡ 1^k = 1 (mod n)`, leaving `c^d ≡ m × 1 = m (mod n)`. Decryption recovers the original message **because** Chapter 8's theorem guarantees it — not by coincidence, and not because the specific numbers above happened to work out.

AN HONEST EDGE CASE

The proof above technically requires `gcd(m, n) = 1`. Testing `m=11` against this exact key (`gcd(11,143)=11`, since 11 is one of the actual prime factors of 143) — the round trip *still* works (`c=132`, decrypts back to `11`). Real RSA is correct for *every* message in `[0,n)`, not just coprime ones, via a more careful argument (the Chinese Remainder Theorem, applied separately mod `p` and mod `q`) — deliberately left out of this course's own stated scope

> (Chapter 1). The Euler's-theorem proof above is complete and honest for the coprime case, which is the overwhelming majority of real messages.

Security: Why Factoring n Breaks Everything

Anyone who can factor the public `n` back into `p` and `q` can recompute `ϕ(n) = (p−1)(q−1)` directly, then run Chapter 5's own extended Euclidean algorithm to compute `d` exactly as the key's own owner did — completely breaking the private key. This is **Chapter 1's own multiply-vs-factor asymmetry**, made concrete: `n` is public precisely because computing it from `p` and `q` is instant, while the reverse — factoring `n` back into `p` and `q` — is (for large enough primes) computationally infeasible with any known method. Every piece of this course has been building toward exactly this one sentence.

Toy RSA in Code

```
def extended_gcd(a, b):
    old_r, r = a, b
    old_s, s = 1, 0
    while r != 0:
        q = old_r // r
        old_r, r = r, old_r - q * r
        old_s, s = s, old_s - q * s
    return old_r, old_s

def generate_keys(p, q, e):
    n = p * q
    phi_n = (p - 1) * (q - 1)
    g, x = extended_gcd(e, phi_n)
    assert g == 1, "e must be coprime with phi(n)"
    d = x % phi_n
    return (n, e), (n, d)   # public key, private key

def encrypt(m, public_key):
    n, e = public_key
    return pow(m, e, n)

def decrypt(c, private_key):
    n, d = private_key
    return pow(c, d, n)

pub, priv = generate_keys(11, 13, 7)
c = encrypt(9, pub)
m = decrypt(c, priv)
print(pub, priv, c, m)   # (143, 7) (143, 103) 48 9
```

Hands-On Exercises

EXERCISE 1

Generate a full RSA key pair for `p=5`, `q=11`, using `e=3`. Show every step (n, φ(n), the gcd check, and d via the extended Euclidean algorithm), then encrypt and decrypt `m=4`, confirming the round trip.

View solution

EXERCISE 2

For `p=7`, `q=13` (so `n=91`, `ϕ(n)=72`), a colleague proposes `e=6`. Explain why this choice is invalid, then choose a valid `e`, compute the corresponding `d`, and encrypt/decrypt `m=6` to confirm it works.

View solution

EXERCISE 3

Using this chapter's own correctness proof, explain in your own words each of the three substitutions that turn `c^d` into `m` — specifically, where `ed = 1 + kϕ(n)` comes from, and where Euler's theorem is actually used in the derivation.

View solution

CHAPTER 9 QUICK REFERENCE

- **Key generation:** pick primes p,q → n=pq → φ(n)=(p−1)(q−1) → choose e coprime to φ(n) → d=e^{-1} mod φ(n)
- **Encrypt:** `c = m^e mod n`. **Decrypt:** `m = c^d mod n` — both are Chapter 7's own fast modular exponentiation
- Correctness is Euler's theorem (Ch.8), applied directly to `ed ≡ 1 (mod ϕ(n))` — not an assumption
- Verified round trip for 6 different messages with the same key pair, all correct
- Security rests entirely on Chapter 1's own multiply-vs-factor asymmetry: computing n from p,q is instant; reversing it isn't
- **Next chapter:** Capstone — building a toy RSA implementation

CHAPTER 10 OF 10

Capstone — Building a Toy RSA Implementation

Number Theory & Cryptographic Math

Chapter 10 · Capstone — Building a Toy RSA Implementation

One continuous project, touching every chapter of this course in the order a real implementation actually needs them: generate real primes, build a real key pair, encrypt and decrypt a real word, measure the real cost of doing it the slow way versus the fast way — then, finally, sit in the attacker's seat and try to break the exact system just built.

STEP	TASK	CHAPTER(S) USED
1	Generate and verify two real primes	Ch.6 (primality testing)
2	Compute the modulus and totient	Ch.2, Ch.8
3	Choose a valid public exponent	Ch.4 (GCD check)
4	Compute the private exponent	Ch.5 (extended Euclidean algorithm)
5-6	Encrypt and decrypt a real word	Ch.3, Ch.7, Ch.8 (correctness)
7	Measure fast vs. naive exponentiation, on this exact key	Ch.7
8	Attempt to break the system: factor n back into p, q	Ch.1 (the core asymmetry), Ch.6

Step 1 — Generate and Verify Two Real Primes

CH.6

Larger primes than Chapter 9's own toy example (`p=11, q=13`), still small enough to trace by hand: candidates `211` and `223` .

VERIFIED DIRECTLY — TRIAL DIVISION

`211` : no divisor found across **13** trial divisions (up to `√211 ≈ 14.5`) — **prime**. `223` : no divisor found across **13** trial divisions (up to `√223 ≈ 14.9`) — **prime**.

Step 2 — Compute the Modulus and Totient

CH.2 · CH.8

$n = 211 \times 223 = 47{,}053$. $\phi(n) = (211-1)(223-1) = 210 \times 222 = 46{,}620$.

Step 3 — Choose a Valid Public Exponent

CH.4

Testing candidates against Chapter 4's own GCD check, $\gcd(e, 46620) = 1$:

THREE CANDIDATES REJECTED — A REAL FINDING, NOT A HYPOTHETICAL

$e=7$: $\gcd(7, 46620) = 7$ — **rejected**. $e=5$: $\gcd(5, 46620) = 5$ — **rejected**. $e=3$: $\gcd(3, 46620) = 3$ — **rejected**. All three fail because $46{,}620$ genuinely is divisible by 3, 5, and 7 — this isn't a contrived example, it's what actually happens when $\phi(n)$ is checked against small, tempting exponent choices.

VERIFIED DIRECTLY

$e=17$: $\gcd(17, 46620) = 1$ — **accepted**.

Step 4 — Compute the Private Exponent

CH.5

Running the extended Euclidean algorithm on $(17, 46620)$:

VERIFIED DIRECTLY

$d = 38{,}393$. Check: $17 \times 38393 \bmod 46620 = 652{,}681 \bmod 46620 = 1$. **Public key: (n=47053, e=17). Private key: (n=47053, d=38393).**

Steps 5-6 — Encrypt and Decrypt a Real Word

CH.3 · CH.7 · CH.8

Encoding `"RSA"` letter by letter (`A=1, B=2, ... Z=26`), encrypting each letter's code independently with `c = m^e mod n` , then decrypting with `m = c^d mod n` :

VERIFIED DIRECTLY

`R=18 → c=25621 → decrypted=18` . `S=19 → c=16608 → decrypted=19` . `A=1 → c=1 → decrypted=1` . Reassembled: **"RSA"** — the exact original word, recovered letter for letter.

AN HONEST EDGE CASE, CAUGHT IN THIS VERY RUN

Notice `A` (code `1`) encrypted to `1` — completely unchanged. This isn't a bug specific to this key: `1^e mod n = 1` for *any* public exponent and *any* modulus, since multiplying `1` by itself never changes it. Real RSA implementations pad messages specifically to avoid ever encrypting a raw, small, predictable value like `1` — a genuine, small weakness this toy version inherits by not bothering with padding, named here rather than swept under the rug.

Step 7 — Fast Exponentiation's Real Payoff, on This Exact Key

CH.7

Chapter 7 proved `O(log b)` beats `O(b)` in the abstract. Here it is, measured on the actual `d=38,393` this project just generated:

VERIFIED DIRECTLY

Decryption (`m^38393 mod n`): naive approach needs **38,393** multiplications; fast exponentiation needs **26**. Encryption (`m^17 mod n`): naive needs **17**; fast needs **7**. This toy key's own private exponent alone would take a naive implementation over a thousand times longer than the fast one to use even once.

Step 8 — The Security Audit: Try to Break What Was Just Built

CH.1 · CH.6

Switching seats: given only the public key `(n=47053, e=17)` — exactly what an attacker would see — how much work does factoring `n` back into `p` and `q` actually take, with no shortcuts?

VERIFIED DIRECTLY — CHAPTER 1'S OWN ASYMMETRY, DEMONSTRATED ON THIS PROJECT'S REAL NUMBERS

Multiplying `211 × 223` to build `n` in Step 2: **one operation**. Trial-dividing `47,053` back down to find its smaller factor, with no prior knowledge: **210 trial divisions**, before the factor `211` is finally found. A **210×** asymmetry — on a modulus barely five digits long. Real RSA moduli are hundreds of digits long, where this exact same asymmetry, scaled up, is what makes factoring genuinely infeasible rather than merely inconvenient.

What This Course Doesn't Cover

As stated honestly back in Chapter 1: a full academic number theory curriculum, elliptic-curve cryptography, the discrete logarithm problem in depth, and post-quantum cryptography were all named as deliberately out of scope, and stayed out of scope through all ten chapters. This course built the specific number-theoretic machinery RSA needs — divisibility, modular arithmetic, GCDs, modular inverses, primality, fast exponentiation, and the totient/Fermat/Euler theorems — not an exhaustive tour of either field.

Where This Course Connects

This course is the mathematical foundation underneath Security's own `crypto1` (Cryptography Fundamentals) and `https1` (HTTPS/TLS Fundamentals), both of which use RSA and public-key cryptography operationally without deriving the number theory behind them — this course is that derivation, made concrete on real (if small) numbers throughout. Algorithms & Complexity's own growth-rate and recursion machinery was used directly in Chapters 4 and 7; Discrete Mathematics Fundamentals' own direct-proof style was the template for Chapter 8's Fermat's Little Theorem proof.

Hands-On Exercises

EXERCISE 1

Verify whether `227` is prime using trial division, showing the search bound and the total number of trial divisions needed.

View solution

EXERCISE 2

Using this chapter's own `ϕ(n)=46,620`, check whether `e=19` is a valid public exponent choice. If it is, compute the corresponding private exponent `d` using the extended Euclidean algorithm, and verify `ed ≡ 1 (mod ϕ(n))`.

View solution

EXERCISE 3

Explain, in your own words, why `m=1` is only one example of a broader category of "predictable" messages a real RSA system needs to guard against — specifically, is `m=0` also a problem? What about a value of `m` equal to the modulus `n` itself? Reason from first principles (what does `m^e mod n` actually compute in each case), not just this chapter's own `m=1` example.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked project:** real verified primes (Ch.6) → n, φ(n) (Ch.2, Ch.8) → a rejected-then-accepted e (Ch.4) → d via extended Euclid (Ch.5) → encrypt/decrypt a real word (Ch.3, Ch.7, Ch.8) → fast-vs-naive cost measured on this exact key (Ch.7) → factor n back to break it (Ch.1, Ch.6)
- Three exponent candidates (3, 5, 7) were genuinely rejected before finding a valid one — a real outcome, not a staged example
- `1^e mod n = 1` always — a real, honestly-named reason production RSA pads messages before encrypting
- Fast exponentiation cut this project's own decryption cost from 38,393 multiplications to 26
- Factoring this project's own 5-digit modulus took 210× the work of building it — the exact asymmetry Chapter 1 opened with, now measured on real numbers from this project
- **Course complete** — Number Theory & Cryptographic Math, 10 chapters, from divisibility to a working (if small) RSA implementation