The Little Book of Algorithms

Version 0.1.0

Duc-Tam Nguyen

2025-09-09

Table of contents

Roadmap

The Little Book of Algorithms is a multi-volume project. Each volume has a clear sequence of chapters, and each chapter has three levels of depth (L0 beginner intuition, L1 practical techniques, L2 advanced systems/theory). This roadmap outlines the plan for development and publication.

Goals

- Establish a consistent layered structure across all chapters.
- Provide runnable implementations in Python, C, Go, Erlang, and Lean.
- Ensure Quarto build supports HTML, PDF, EPUB, and LaTeX.
- Deliver both pedagogy (L0) and production insights (L2).

Volumes

Volume I - Structures Linéaires

- Chapter 0 - Foundations
- Chapter 1 - Numbers
- Chapter 2 - Arrays
- Chapter 3 - Strings
- Chapter 4 - Linked Lists
- Chapter 5 - Stacks & Queues

Volume II - Algorithmes Fondamentaux

- Chapter 6 - Searching
- Chapter 7 - Selection
- Chapter 8 - Sorting
- Chapter 9 - Amortized Analysis

Volume III - Structures Hiérarchiques

- Chapter 10 - Tree Fundamentals
- Chapter 11 - Heaps & Priority Queues
- Chapter 12 - Binary Search Trees
- Chapter 13 - Balanced Trees & Ordered Maps
- Chapter 14 - Range Queries
- Chapter 15 - Vector Databases

Volume IV - Paradigmes Algorithmiques

- Chapter 16 - Divide-and-Conquer
- Chapter 17 - Greedy
- Chapter 18 - Dynamic Programming
- Chapter 19 - Backtracking & Search

Volume V - Graphes et Complexité

- Chapter 20 - Graph Basics
- Chapter 21 - DAGs & SCC
- Chapter 22 - Shortest Paths
- Chapter 23 - Flows & Matchings
- Chapter 24 - Tree Algorithms
- Chapter 25 - Complexity & Limits
- Chapter 26 - External & Cache-Oblivious
- Chapter 27 - Probabilistic & Streaming
- Chapter 28 - Engineering

Milestones

1. Complete detailed outlines for all chapters (L0, L1, L2).
2. Write draft text for all L0 sections (intuition, analogies, simple examples).
3. Expand each chapter with L1 content (implementations, correctness arguments, exercises).
4. Add L2 content (systems insights, proofs, optimizations, advanced references).
5. Develop and test runnable code in `src/` across Python, C, Go, Erlang, and Lean.
6. Integrate diagrams, figures, and visual explanations.
7. Finalize Quarto build setup for HTML, PDF, and EPUB.
8. Release first public edition (HTML + PDF).
9. Add LaTeX build, refine EPUB, and polish cross-references.
10. Publish on GitHub Pages and archive DOI.

11. Gather feedback, refine explanations, and expand exercises/problem sets.
12. Long-term: maintain as a living reference with continuous updates and companion volumes.

Deliverables

- Quarto project with 29 chapters (00–28).
- Multi-language reference implementations.
- Learning matrix in README for navigation.
- ROADMAP.md (this file) to track progress.

Long-term Vision

- Maintain the repository as a living reference.
- Extend with exercises, problem sets, and quizzes.
- Build a dependency map across volumes for prerequisites.
- Connect to companion "Little Book" series (linear algebra, calculus, probability).

Chapter 1. Numbers

1.1 Representation

1.1 L0. Decimal and Binary Basics

A number representation is a way of writing numbers using symbols and positional rules. Humans typically use decimal notation, while computers rely on binary because it aligns with the two-state nature of electronic circuits. Understanding both systems is the first step in connecting mathematical intuition with machine computation.

Numbers in Everyday Life

Humans work with the decimal system (base 10), which uses digits 0 through 9. Each position in a number has a place value that is a power of 10.

$$427 = 4 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$$

This principle of *positional notation* is the same idea used in other bases.

Numbers in Computers

Computers, however, operate in binary (base 2). A binary digit (bit) can only be 0 or 1, matching the two stable states of electronic circuits (off/on). Each binary place value represents a power of 2.

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11_{10}$$

Just like in decimal where $9 + 1 = 10$, in binary $1 + 1 = 10_2$.

Conversion Between Decimal and Binary

To convert from decimal to binary, repeatedly divide the number by 2 and record the remainders. Then read the remainders from bottom to top.

Example: Convert 42_{10} into binary.

- $42 \div 2 = 21$ remainder 0
- $21 \div 2 = 10$ remainder 1
- $10 \div 2 = 5$ remainder 0
- $5 \div 2 = 2$ remainder 1
- $2 \div 2 = 1$ remainder 0
- $1 \div 2 = 0$ remainder 1

Reading upward: 101010_2.

To convert from binary to decimal, expand into powers of 2 and sum:

$$101010_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 42_{10}$$

Worked Example (Python)

```
n = 42
print("Decimal:", n)
print("Binary :", bin(n))   # 0b101010

# binary literal in Python
b = 0b101010
print("Binary literal:", b)

# converting binary string to decimal
print("From binary '1011':", int("1011", 2))
```

Output:

```
Decimal: 42
Binary : 0b101010
Binary literal: 42
From binary '1011': 11
```

Why It Matters

- All information inside a computer — numbers, text, images, programs — reduces to binary representation.
- Decimal and binary conversions are the first bridge between human-friendly math and machine-level data.
- Understanding binary is essential for debugging, low-level programming, and algorithms that depend on bit operations.

Exercises

1. Write the decimal number 19 in binary.
2. Convert the binary number `10101` into decimal.
3. Show the repeated division steps to convert 27 into binary.
4. Verify in Python that `0b111111` equals 63.
5. Explain why computers use binary instead of decimal.

1.1 L1. Beyond Binary: Octal, Hex, and Two's Complement

Numbers are not always written in base-10 or even in base-2. For efficiency and compactness, programmers often use octal (base-8) and hexadecimal (base-16). At the same time, negative numbers must be represented reliably; modern computers use two's complement for this purpose.

Octal and Hexadecimal

Octal and hex are simply alternate numeral systems.

- Octal (base 8): digits `0`–`7`.
- Hexadecimal (base 16): digits `0`–`9` plus `A`–`F`.

Why they matter:

- Hex is concise: one hex digit = 4 binary bits.
- Octal was historically convenient: one octal digit = 3 binary bits (useful on early 12-, 24-, or 36-bit machines).

For example, the number 42 is written as:

Decimal	Binary	Octal	Hex
42	101010	52	2A

Two's Complement

To represent negative numbers, we cannot just "stick a minus sign" in memory. Instead, binary uses two's complement:

1. Choose a fixed bit-width (say 8 bits).
2. For a negative number `-x`, compute `2^bits - x`.
3. Store the result as an ordinary binary integer.

Example with 8 bits:

- `+5` → `00000101`
- `-5` → `11111011`
- `-1` → `11111111`

Why two's complement is powerful:

- Addition and subtraction "just work" with the same circuitry for signed and unsigned.
- There is only one representation of zero.

Working Example (Python)

```
# Decimal 42 in different bases
n = 42
print("Decimal:", n)
print("Binary :", bin(n))
print("Octal  :", oct(n))
print("Hex    :", hex(n))

# Two's complement for -5 in 8 bits
def to_twos_complement(x: int, bits: int = 8) -> str:
    if x >= 0:
        return format(x, f"0{bits}b")
    return format((1 << bits) + x, f"0{bits}b")

print("+5:", to_twos_complement(5, 8))
print("-5:", to_twos_complement(-5, 8))
```

Output:

```
Decimal: 42
Binary : 0b101010
Octal  : 0o52
Hex    : 0x2a
+5: 00000101
-5: 11111011
```

Why It Matters

- Programmer convenience: Hex makes binary compact and human-readable.
- Hardware design: Two's complement ensures arithmetic circuits are simple and unified.
- Debugging: Memory dumps, CPU registers, and network packets are usually shown in hex.

Exercises

1. Convert `100` into binary, octal, and hex.
2. Write `-7` in 8-bit two's complement.
3. Verify that `0xFF` is equal to 255.
4. Parse the bitstring `"11111001"` as an 8-bit two's complement number.
5. Explain why engineers prefer two's complement over "sign-magnitude" representation.

1.1 L2. Floating-Point and Precision Issues

Not all numbers are integers. To approximate fractions, scientific notation, and very large or very small values, computers use floating-point representation. The de-facto standard is IEEE-754, which defines how real numbers are encoded, how special values are handled, and what precision guarantees exist.

Structure of Floating-Point Numbers

A floating-point value is composed of three fields:

1. Sign bit (s) — indicates positive (`0`) or negative (`1`).
2. Exponent (e) — determines the scale or "magnitude."
3. Mantissa / significand (m) — contains the significant digits.

The value is interpreted as:

$$(-1)^s \times 1.m \times 2^{(e-\text{bias})}$$

Example: IEEE-754 single precision (32 bits)

- 1 sign bit
- 8 exponent bits (bias = 127)
- 23 mantissa bits

Exact vs Approximate Representation

Some numbers are represented exactly:

- `1.0` has a clean binary form.

Others cannot be represented precisely:

- `0.1` in decimal is a repeating fraction in binary, so the closest approximation is stored.

Python example:

```
a = 0.1 + 0.2
print("0.1 + 0.2 =", a)
print("Equal to 0.3?", a == 0.3)
```

Output:

```
0.1 + 0.2 = 0.30000000000000004
Equal to 0.3? False
```

Special Values

IEEE-754 reserves encodings for special cases:

Sign	Exponent	Mantissa	Meaning
0/1	all 1s	0	$+\infty$ / $-\infty$
0/1	all 1s	nonzero	NaN (Not a Number)
0/1	all 0s	nonzero	Denormals (gradual underflow)

Examples:

- Division by zero produces infinity: `1.0 / 0.0 = inf`.
- `0.0 / 0.0` yields `NaN`, which propagates in computations.
- Denormals allow gradual precision near zero.

Arbitrary Precision

Languages like Python and libraries like GMP provide arbitrary-precision arithmetic:

- Integers (`int`) can grow as large as memory allows.
- Decimal libraries (`decimal.Decimal` in Python) allow exact decimal arithmetic.
- These are slower, but essential for cryptography, symbolic computation, and finance.

Worked Example (Python)

```
import math

print("Infinity:", 1.0 / 0.0)
print("NaN:", 0.0 / 0.0)

print("Is NaN?", math.isnan(float('nan')))
print("Is Inf?", math.isinf(float('inf')))

# Arbitrary precision integer
big = 2200
print("2200 =", big)
```

Why It Matters

- Rounding surprises: Many decimal fractions cannot be represented exactly.
- Error propagation: Repeated arithmetic may accumulate tiny inaccuracies.
- Special values: NaN and infinity must be handled carefully.
- Domain correctness: Cryptography, finance, and symbolic algebra require exact precision.

Exercises

1. Write down the IEEE-754 representation (sign, exponent, mantissa) of `1.0`.
2. Explain why `0.1` is not exactly representable in binary.
3. Test in Python whether `float('nan') == float('nan')`. What happens, and why?
4. Find the smallest positive number you can add to `1.0` before it changes (machine epsilon).
5. Why is arbitrary precision slower but critical in some applications?

1.2 Basic Operations

1.2 L0. Addition, Subtraction, Multiplication, Division

An arithmetic operation combines numbers to produce a new number. At this level we focus on four basics: addition, subtraction, multiplication, and division—first with decimal intuition, then a peek at how the same ideas look in binary. Mastering these is essential before moving to algorithms that build on them.

Intuition: place value + carrying/borrowing

All four operations are versions of combining place values (ones, tens, hundreds …; or in binary: ones, twos, fours …).

- Addition: add column by column; if a column exceeds the base, carry 1 to the next column.
- Subtraction: subtract column by column; if a column is too small, borrow 1 from the next column.
- Multiplication: repeated addition; multiply by each digit and shift (place value), then add partial results.
- Division: repeated subtraction or sharing; find how many times a number "fits," track the remainder.

These rules are identical in any base. Only the place values change.

Decimal examples (by hand)

1. Addition (carry)

```
  478
+ 259
----
  737    (8+9=17 → write 7, carry 1; 7+5+1=13 → write 3, carry 1; 4+2+1=7)
```

2. Subtraction (borrow)

```
  503
-  78
----
  425    (3-8 borrow → 13-8=5; 0 became -1 so borrow from 5 → 9-7=2; 4 stays 4)
```

3. Multiplication (partial sums)

```
  214
×   3
----
  642    (214*3 = 642)
```

4. Long division (quotient + remainder)

```
 47 ÷ 5 → 9 remainder 2   (because 5*9 = 45, leftover 2)
```

Binary peek (same rules, base 2)

- Add rules: 0+0=0, 0+1=1, 1+0=1, 1+1=10 (write 0, carry 1)
- Subtract rules: 0−0=0, 1−0=1, 1−1=0, 0−1 → borrow (becomes 10 −1=1, borrow 1)

Example: $1011_2 + 0110_2$

```
  1011
+ 0110
------
 10001   (1+0=1; 1+1=0 carry1; 0+1+carry=0 carry1; 1+0+carry=0 carry1 → carry out)
```

Worked examples (Python)

```python
# Basic arithmetic with integers
a, b = 478, 259
print("a+b =", a + b)      # 737
print("a-b =", a - b)      # 219
print("a*b =", a * b)      # 123,  478*259 = 123,  ... actually compute:
print("47//5 =", 47 // 5)  # integer division -> 9
print("47%5  =", 47 % 5)   # remainder -> 2

# Show carry/borrow intuition using binary strings
x, y = 0b1011, 0b0110
s = x + y
print("x+y (binary):", bin(x), "+", bin(y), "=", bin(s))

# Small helper: manual long division that returns (quotient, remainder)
def long_divide(n: int, d: int):
    if d == 0:
```

```
        raise ZeroDivisionError("division by zero")
    q = n // d
    r = n % d
    return q, r

print("long_divide(47,5):", long_divide(47, 5))  # (9, 2)
```

Note: `//` is integer division in Python; `%` is the remainder. For now we focus on integers (no decimals).

Why it matters

- Every higher-level algorithm (searching, hashing, cryptography, numeric methods) relies on these operations.
- Understanding carry/borrow makes binary arithmetic and bit-level reasoning feel natural.
- Knowing integer division and remainder is vital for base conversions, hashing (`mod`), and many algorithmic patterns.

Exercises

1. Compute by hand, then verify in Python:
   - $326 + 589$
   - $704 - 259$
   - 38×12
   - $123 \div 7$ (give quotient and remainder)
2. In binary, add $10101_2 + 111_{(2)}$. Show carries.
3. Write a short Python snippet that prints the quotient and remainder for `n=200` divided by `d=23`.
4. Convert your remainder into a sentence: "200 = 23 × (quotient) + (remainder)".
5. Challenge: Multiply 19×23 by hand using partial sums; then check with Python.

1.2 L1. Division, Modulo, and Efficiency

Beyond the simple four arithmetic operations, programmers need to think about division with remainder, the modulo operator, and how efficient these operations are on real machines. Addition and subtraction are almost always "constant time," but division can be slower, and understanding modulo is essential for algorithms like hashing, cryptography, and scheduling.

Integer Division and Modulo

For integers, division produces both a quotient and a remainder.

- Mathematical definition: for integers n, d with $d \neq 0$,

$$n = d \times q + r, \quad 0 \leq r < |d|$$

  where q is the quotient, r the remainder.

- Programming notation (Python):
  - `n // d` → quotient
  - `n % d` → remainder

Examples:

- `47 // 5 = 9`, `47 % 5 = 2` because $47 = 5 \times 9 + 2$.
- `23 // 7 = 3`, `23 % 7 = 2` because $23 = 7 \times 3 + 2$.

n	d	n // d	n % d
47	5	9	2
23	7	3	2
100	9	11	1

Modulo in Algorithms

The modulo operation is a workhorse in programming:

- Hashing: To map a large integer into a table of size `m`, use `key % m`.
- Cyclic behavior: To loop back after `7` days in a week: `(day + shift) % 7`.
- Cryptography: Modular arithmetic underlies RSA, Diffie–Hellman, and many number-theoretic algorithms.

Efficiency Considerations

- Addition and subtraction: generally 1 CPU cycle.
- Multiplication: slightly more expensive, but still fast on modern hardware.
- Division and modulo: slower, often an order of magnitude more costly than multiplication.

Practical tricks:

- If `d` is a power of two, `n % d` can be computed by a bitmask.
  - Example: `n % 8 == n & 7` (since $8 = 2^3$).
- Some compilers automatically optimize modulo when the divisor is constant.

Worked Example (Python)

```python
# Quotient and remainder
n, d = 47, 5
print("Quotient:", n // d)  # 9
print("Remainder:", n % d)  # 2

# Identity check: n == d*q + r
q, r = divmod(n, d)  # built-in tuple return
print("Check:", d*q + r == n)

# Modulo for cyclic behavior: days of week
days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
start = 5  # Saturday
shift = 4
future_day = days[(start + shift) % 7]
print("Start Saturday + 4 days =", future_day)

# Optimization: power-of-two modulo with bitmask
for n in [5, 12, 20]:
    print(f"{n} % 8 = {n % 8}, bitmask {n & 7}")
```

Output:

```
Quotient: 9
Remainder: 2
Check: True
Start Saturday + 4 days = Wed
5 % 8 = 5, bitmask 5
12 % 8 = 4, bitmask 4
20 % 8 = 4, bitmask 4
```

Why It Matters

- Real programs rely heavily on modulo for indexing, hashing, and wrap-around logic.
- Division is computationally more expensive; knowing when to replace it with bit-level operations improves performance.
- Modular arithmetic introduces a new "world" where numbers wrap around — the foundation of many advanced algorithms.

Exercises

1. Compute by hand and confirm in Python:
   - `100 // 9` and `100 % 9`
   - `123 // 11` and `123 % 11`
2. Write a function that simulates a clock: given `hour` and `shift`, return the new hour (24-hour cycle).
3. Prove the identity: for any integers `n` and `d`,

   ```
   n == d * (n // d) + (n % d)
   ```

   by trying with random values.
4. Show how to replace `n % 16` with a bitwise operation. Why does it work?
5. Challenge: Write a short Python function to check if a number is divisible by 7 using only `%` and `//`.

1.2 L2. Fast Arithmetic Algorithms

When numbers grow large, the naïve methods for multiplication and division become too slow. On paper, long multiplication takes $O(n^2)$ steps for n-digit numbers. Computers face the same issue: multiplying two very large integers digit by digit can be expensive. Fast arithmetic algorithms reduce this cost, using clever divide-and-conquer techniques or transformations into other domains.

Multiplication Beyond the School Method

Naïve long multiplication

- Treats an n-digit number as a sequence of digits.
- Each digit of one number multiplies every digit of the other.
- Complexity: $O(n^2)$.
- Works fine for small integers, but too slow for cryptography or big-number libraries.

Karatsuba's Algorithm

- Discovered in 1960 by Anatoly Karatsuba.
- Idea: split numbers into halves and reduce multiplications.
- Complexity: $O(n^{\log_2 3}) \approx O(n^{1.585})$.
- Recursive strategy:
  - For numbers $x = x_1 \cdot B^m + x_0$, $y = y_1 \cdot B^m + y_0$.
  - Compute 3 multiplications instead of 4:
    * $z_0 = x_0 y_0$
    * $z_2 = x_1 y_1$
    * $z_1 = (x_0 + x_1)(y_0 + y_1) - z_0 - z_2$
  - Result: $z_2 \cdot B^{2m} + z_1 \cdot B^m + z_0$.

FFT-based Multiplication (Schönhage–Strassen and successors)

- Represent numbers as polynomials of their digits.
- Multiply polynomials efficiently using Fast Fourier Transform.
- Complexity: near $O(n \log n)$.
- Used in modern big-integer libraries (e.g. GNU MP, Java's `BigInteger`).

Division Beyond Long Division

- Naïve long division: $O(n^2)$ for n-digit dividend.
- Newton's method for reciprocal: approximate $1/d$ using Newton–Raphson iterations, then multiply by n.
- Complexity: tied to multiplication — if multiplication is fast, so is division.

Modular Exponentiation

Fast arithmetic also matters in modular contexts (cryptography).

- Compute $a^b \bmod m$ efficiently.
- Square-and-multiply (binary exponentiation):
  - Write b in binary.
  - For each bit: square result, multiply if bit=1.
  - Complexity: $O(\log b)$ multiplications.

Worked Example (Python)

```
# Naïve multiplication
def naive_mul(x: int, y: int) -> int:
    return x * y  # Python already uses fast methods internally

# Karatsuba multiplication (recursive, simplified)
def karatsuba(x: int, y: int) -> int:
    # base case
    if x < 10 or y < 10:
        return x * y
    # split numbers
    n = max(x.bit_length(), y.bit_length())
    m = n // 2
    high1, low1 = divmod(x, 1 << m)
    high2, low2 = divmod(y, 1 << m)
    z0 = karatsuba(low1, low2)
    z2 = karatsuba(high1, high2)
    z1 = karatsuba(low1 + high1, low2 + high2) - z0 - z2
    return (z2 << (2*m)) + (z1 << m) + z0

# Modular exponentiation (square-and-multiply)
```

```
def modexp(a: int, b: int, m: int) -> int:
    result = 1
    base = a % m
    exp = b
    while exp > 0:
        if exp & 1:
            result = (result * base) % m
        base = (base * base) % m
        exp >>= 1
    return result

# Demo
print("Karatsuba(1234, 5678) =", karatsuba(1234, 5678))
print("pow(7, 128, 13) =", modexp(7, 128, 13))  # fast modular exponentiation
```

Output:

```
Karatsuba(1234, 5678) = 7006652
pow(7, 128, 13) = 3
```

Why It Matters

- Cryptography: RSA requires multiplying and dividing integers with thousands of digits.
- Computer algebra systems: symbolic computation depends on fast polynomial/integer arithmetic.
- Big data / simulation: arbitrary precision needed when floats are not exact.

Exercises

1. Multiply 31415926 × 27182818 using:
    - Python's `*`
    - Your Karatsuba implementation. Compare results.
2. Implement `modexp(a, b, m)` for $a = 5, b = 117, m = 19$. Confirm with Python's built-in `pow(a, b, m)`.
3. Explain why Newton's method for division depends on fast multiplication.
4. Research: what is the current fastest known multiplication algorithm for large integers?
5. Challenge: Modify Karatsuba to print intermediate `z0, z1, z2` values for small inputs to visualize the recursion.

1.3 Properties

1.3 L0 — Simple Number Properties

Numbers have patterns that help us reason about algorithms without heavy mathematics. At this level we focus on basic properties: even vs odd, divisibility, and remainders. These ideas show up everywhere—from loop counters to data structure layouts.

Even and Odd

A number is even if it ends with digit 0, 2, 4, 6, or 8 in decimal, and odd otherwise.

- In binary, checking parity is even easier: the last bit tells the story.
  - …0 → even
  - …1 → odd

Example in Python:

```
def is_even(n: int) -> bool:
    return n % 2 == 0

print(is_even(10))  # True
print(is_even(7))   # False
```

Divisibility

We often ask: does one number divide another?

- `a` is divisible by `b` if there exists some integer `k` with `a = b * k`.
- In code: `a % b == 0`.

Examples:

- 12 is divisible by 3 → `12 % 3 == 0`.
- 14 is not divisible by 5 → `14 % 5 == 4`.

Remainders and Modular Thinking

When dividing, the remainder is what's left over.

- Example: `17 // 5 = 3`, remainder `2`.
- Modular arithmetic wraps around like a clock:
  - `(17 % 5) = 2` → same as "2 o'clock after going 17 steps around a 5-hour clock."

This "wrap-around" view is central in array indexing, hashing, and cryptography later on.

Why It Matters

- Algorithms: Parity checks decide branching (e.g., even-odd optimizations).
- Data structures: Array indices often wrap around using `%`.
- Everyday: Calendars cycle days of the week; remainders formalize that.

Exercises

1. Write a function that returns `"even"` or `"odd"` for a given number.
2. Check if 91 is divisible by 7.
3. Compute the remainder of 100 divided by 9.
4. Use `%` to simulate a 7-day week: if today is day 5 (Saturday) and you add 10 days, what day is it?
5. Find the last digit of `2^15` without computing the full number (hint: check the remainder mod 10).

1.3 L1 — Classical Number Theory Tools

Beyond simple parity and divisibility, algorithms often need deeper number properties. At this level we introduce a few "toolkit" ideas from elementary number theory: greatest common divisor (GCD), least common multiple (LCM), and modular arithmetic identities. These are lightweight but powerful concepts that show up in algorithm design, cryptography, and optimization.

Greatest Common Divisor (GCD)

The GCD of two numbers is the largest number that divides both.

- Example: `gcd(20, 14) = 2`.
- Why useful: GCD simplifies fractions, ensures ratios are reduced, and appears in algorithm correctness proofs.

Euclid's Algorithm: Instead of trial division, we can compute GCD quickly:

```
gcd(a, b) = gcd(b, a % b)
```

This repeats until `b = 0`, at which point `a` is the answer.

Python example:

```python
def gcd(a: int, b: int) -> int:
    while b:
        a, b = b, a % b
    return a

print(gcd(20, 14))  # 2
```

Least Common Multiple (LCM)

The LCM of two numbers is the smallest positive number divisible by both.

- Example: `lcm(12, 18) = 36`.
- Connection to GCD:

  ```
  lcm(a, b) = (a * b) // gcd(a, b)
  ```

This is useful in scheduling, periodic tasks, and synchronization problems.

Modular Arithmetic Identities

Remainders behave predictably under operations:

- Addition: `(a + b) % m = ((a % m) + (b % m)) % m`
- Multiplication: `(a * b) % m = ((a % m) * (b % m)) % m`

Example:

- `(123 + 456) % 7 = (123 % 7 + 456 % 7) % 7`
- This property lets us work with small remainders instead of huge numbers, key in cryptography and hashing.

Why It Matters

- Algorithms: GCD ensures efficiency in fraction reduction, graph algorithms, and number-theoretic algorithms.
- Systems: LCM models periodicity, e.g., aligning CPU scheduling intervals.
- Cryptography: Modular arithmetic underpins secure communication (RSA, Diffie-Hellman).
- Practical programming: Modular identities simplify computations with limited ranges (hash tables, cyclic arrays).

Exercises

1. Compute `gcd(252, 198)` by hand using Euclid's algorithm.
2. Write a function that returns the LCM of two numbers. Test it on (12, 18).
3. Show that `(37 + 85) % 12` equals `((37 % 12) + (85 % 12)) % 12`.
4. Reduce the fraction `84/126` using GCD.
5. Find the smallest day `d` such that `d` is a multiple of both 12 and 18 (hint: LCM).

1.3 L2 — Advanced Number Theory in Algorithms

At this level, we move beyond everyday divisibility and Euclid's algorithm. Modern algorithms frequently rely on deep number theory to achieve efficiency. Topics such as modular inverses, Euler's totient function, and primality tests are crucial foundations for cryptography, randomized algorithms, and competitive programming.

Modular Inverses

The modular inverse of a number `a` (mod `m`) is an integer `x` such that:

```
(a * x) % m = 1
```

- Example: the inverse of 3 modulo 7 is 5, because `(3*5) % 7 = 15 % 7 = 1`.
- Existence: an inverse exists if and only if `gcd(a, m) = 1`.
- Computation: using the Extended Euclidean Algorithm.

This is the backbone of modular division and is heavily used in cryptography (RSA), hash functions, and matrix inverses mod `p`.

Euler's Totient Function ( )

The function `(n)` counts the number of integers between 1 and `n` that are coprime to `n`.

- Example: `(9) = 6` because {1, 2, 4, 5, 7, 8} are coprime to 9.
- Key property (Euler's theorem):

  ```
  a^ (n)   1 (mod n)      if gcd(a, n) = 1
  ```

- Special case: Fermat's Little Theorem — for prime `p`,

  ```
  a^(p-1)   1 (mod p)
  ```

This result is central in modular exponentiation and cryptosystems like RSA.

Primality Testing

Determining if a number is prime is easy for small inputs but hard for large ones. Efficient algorithms are essential:

- Trial division: works only for small `n`.
- Fermat primality test: uses Fermat's Little Theorem to detect composites, but can be fooled by Carmichael numbers.
- Miller–Rabin test: a probabilistic algorithm widely used in practice (cryptographic key generation).
- AKS primality test: a deterministic polynomial-time method (theoretical importance).

Example intuition:

- For large `n`, we don't check all divisors; we test properties of `a^k mod n` for random bases `a`.

Why It Matters

- Cryptography: Public-key systems depend on modular inverses, Euler's theorem, and large primes.
- Algorithms: Modular inverses simplify solving equations in modular arithmetic (e.g., Chinese Remainder Theorem applications).
- Practical Computing: Randomized primality tests (like Miller–Rabin) balance correctness and efficiency in real-world systems.

Exercises

1. Find the modular inverse of 7 modulo 13.
2. Compute (10) and verify Euler's theorem for `a = 3`.
3. Use Fermat's test to check whether 341 is prime. (Hint: try `a = 2`.)
4. Implement modular inverse using the Extended Euclidean Algorithm.
5. Research: why do cryptographic protocols prefer Miller–Rabin over AKS, even though AKS is deterministic?

1.4 Overflow & Precision

1.4 L0 - When Numbers Get Too Big or Too Small

Numbers inside a computer are stored with a fixed number of bits. This means they can only represent values up to a certain limit. If a calculation produces a result larger than this limit, the value "wraps around," much like the digits on an odometer rolling over after 999 to 000. This phenomenon is called overflow. Similarly, computers often cannot represent all decimal fractions exactly, leading to tiny errors called precision loss.

Deep Dive

1. Integer Overflow

    - A computer uses a fixed number of bits (commonly 8, 16, 32, or 64) to store integers.
    - An 8-bit unsigned integer can represent values from 0 to 255. Adding 1 to 255 causes the value to wrap back to 0.
    - Signed integers use *two's complement* representation. For an 8-bit signed integer, the range is -128 to $+127$. Adding 1 to 127 makes it overflow to -128.

    Example in binary:

```
11111111  (255) + 1 = 00000000  (0)
01111111  (+127) + 1 = 10000000  (-128)
```

2. Floating-Point Precision

   - Decimal fractions like 0.1 cannot always be represented exactly in binary.
   - As a result, calculations may accumulate tiny errors.
   - For example, repeatedly adding 0.1 may not exactly equal 1.0 due to precision limits.

Example

```python
# Integer overflow simulation with 8-bit values
def add_8bit(a, b):
    result = (a + b) % 256  # simulate wraparound
    return result

print(add_8bit(250, 10))   # 260 wraps to 4
print(add_8bit(255, 1))    # wraps to 0

# Floating-point precision issue
x = 0.1 + 0.2
print(x)           # Expected 0.3, but gives 0.30000000000000004
print(x == 0.3)    # False
```

Why It Matters

- Unexpected results: A calculation may suddenly produce a negative number or wrap around to zero.
- Real-world impact:
  - Video games may show scores jumping strangely if counters overflow.
  - Banking or financial systems must avoid losing cents due to floating-point errors.
  - Engineers and scientists rely on careful handling of precision to ensure correct simulations.
- Foundation for algorithms: Understanding overflow and precision prepares you for later topics like hashing, cryptography, and numerical analysis.

Exercises

1. Simulate a 4-bit unsigned integer system. What happens if you start at 14 and keep adding 1?
2. In Python, try adding `0.1` to itself ten times. Does it equal exactly `1.0`? Why or why not?
3. Write a function that checks if an 8-bit signed integer addition would overflow.
4. Imagine you are programming a digital clock that uses 2 digits for minutes (00–59). What happens when the value goes from 59 to 60? How would you handle this?

1.4 L1 - Detecting and Managing Overflow in Real Programs

Computers don't do math in the abstract. Integers live in fixed-width registers; floats follow IEEE-754. Robust software accounts for these limits up front: choose the right representation, detect overflow, and compare floats safely. The following sections explain how these issues show up in practice and how to design around them.

Deep Dive

1) Integer arithmetic in practice

Fixed width means wraparound at 2^n. Unsigned wrap is modular arithmetic; signed overflow (two's complement) can flip signs. Developers often discover this the hard way when a counter suddenly goes negative or wraps back to zero in production logs.

Bit width & ranges This table reminds us of the hard limits baked into hardware. Once the range is exceeded, the value doesn't "grow bigger"—it wraps.

Width	Signed range	Unsigned range
32	−2,147,483,648 … 2,147,483,647	0 … 4,294,967,295
64	−9,223,372,036,854,775,808 … 9,223,372,036,854,775,807	0 … 18,446,744,073,709,551,615

Overflow semantics by language Each language makes slightly different promises. This matters if you're writing cross-language services or reading binary data across APIs.

Language	Signed overflow	Unsigned overflow	Notes
C/C++	UB (undefined)	Modular wrap	Use UBSan/`-fsanitize=undefined`; widen types or check before add.
Rust	Traps in debug; defined APIs	`wrapping_add`, `checked_add`, `saturating_add`	Make intent explicit.
Java/Kotlin	Wraps (two's complement)	N/A (only signed types)	Use `Math.addExact` to trap.
C#	Wraps by default; `checked` to trap	`checked`/`unchecked` blocks	`decimal` type for money.
Python	Arbitrary precision	Arbitrary precision	Simulates fixed width if needed.

A quick lesson: "wrap" may be safe in crypto or hashing, but it's usually a bug in counters or indices. Always decide what you want up front.

2) Floating-point you can depend on

IEEE-754 doubles have ~15–16 decimal digits and huge dynamic range, but not exact decimal fractions. Think of floats as *convenient approximations*. They are perfect for physics simulations, but brittle when used to represent cents in a bank account.

Where precision is lost These examples show why "0.1 + 0.2 != 0.3" isn't a joke—it's a direct consequence of binary storage.

- Scale mismatch: $1e16 + 1 = 1e16$. The tiny `+1` gets lost.
- Cancellation: subtracting nearly equal numbers deletes significant digits.
- Decimal fractions (`0.1`) are repeating in binary.

Comparing floats Never compare with `==`. Instead, use a mixed relative + absolute check:

$$|x - y| \le \max(\text{rel} \cdot \max(|x|, |y|),\ \text{abs})$$

This makes comparisons robust whether you're near zero or far away.

Rounding modes (when you explicitly care) Most of the time you don't think about rounding—hardware defaults to "round to nearest, ties to even." But when writing financial systems or interval arithmetic, you want to control it.

Mode	Typical use
Round to nearest, ties to even (default)	General numeric work; minimizes bias
Toward 0 / $\pm\infty$	Bounds, interval arithmetic, conservative estimates

Having explicit rounding modes is like having a steering wheel—you don't always turn, but you're glad it's there when the road curves.

Summation strategies The order of addition matters for floats. These options give you a menu of accuracy vs. speed.

Method	Error	Cost	When to use
Naïve left-to-right	Worst	Low	Never for sensitive sums
Pairwise / tree	Better	Med	Parallel reductions, "good default"
Kahan (compensated)	Best	Higher	Financial-ish aggregates, small vectors

You don't need Kahan everywhere, but knowing it exists keeps you from blaming "mystery bugs" on hardware.

Representation choices Sometimes the best answer is to avoid floats entirely. Money is the classic example.

Use case	Recommended representation
Currency, invoicing	Fixed-point (e.g., cents as `int64`) or `decimal`/`BigDecimal`
Scientific compute	`float64`, compensated sums, stable algorithms
IDs, counters	`uint64`/`int64`, detect overflow on boundaries

Code (Python—portable patterns)

```
# 32-bit checked add (raises on overflow)
def add_i32_checked(a: int, b: int) -> int:
    s = a + b
    if s < -2_147_483_648 or s > 2_147_483_647:
        raise OverflowError("int32 overflow")
    return s

# Simulate 32-bit wrap (intentional modular arithmetic)
def add_i32_wrapping(a: int, b: int) -> int:
    s = (a + b) & 0xFFFFFFFF
```

```
    return s - 0x100000000 if s & 0x80000000 else s

# Relative+absolute epsilon float compare
def almost_equal(x: float, y: float, rel=1e-12, abs_=1e-12) -> bool:
    return abs(x - y) <= max(rel * max(abs(x), abs(y)), abs_)

# Kahan (compensated) summation
def kahan_sum(xs):
    s = 0.0
    c = 0.0
    for x in xs:
        y = x - c
        t = s + y
        c = (t - s) - y
        s = t
    return s

# Fixed-point cents (safe for ~±9e16 cents with int64)
def dollars_to_cents(d: str) -> int:
    whole, _, frac = d.partition(".")
    frac = (frac + "00")[:2]
    return int(whole) * 100 + int(frac)

def cents_to_dollars(c: int) -> str:
    sign = "-" if c < 0 else ""
    c = abs(c)
    return f"{sign}{c//100}.{c%100:02d}"
```

These examples are in Python for clarity, but the same ideas exist in every major language.

Why it matters

- Reliability: Silent wrap or float drift becomes data corruption under load or over time.
- Interoperability: Services in different languages disagree on overflow; define and document your contracts.
- Reproducibility: Deterministic numerics (same inputs → same bits) depend on summation order, rounding, and libraries.
- Security: UB-triggered overflows can turn into exploitable states.

This is why “it worked on my laptop” is not enough. You want to be sure it works on every platform, every time.

Exercises

1. Overflow policy: For a metrics pipeline, decide where to use `checked`, `wrapping`, and `saturating` addition—and justify each with failure modes.
2. ULP probe: Find the smallest ϵ such that `1.0 +  != 1.0` in your language; explain how it relates to machine epsilon.
3. Summation bake-off: Sum the first 1M terms of the harmonic series with naïve, pairwise, and Kahan methods; compare results and timings.
4. Fixed-point ledger: Implement deposit/transfer/withdraw using `int64` cents; prove no rounding loss for two-decimal currencies.
5. Boundary tests: Write property tests that `add_i32_checked` raises on `{INT_MAX,1}` and `{INT_MIN,-1}`, and equals modular add where documented.
6. Cross-lang contract: Specify a JSON schema for monetary amounts and counters that avoids float types; include examples and edge cases.

Great — let's rework 1.4 Overflow & Precision (L2) into a friendlier deep dive, using the same pattern: structured sections, tables, and added "bridge" sentences that guide the reader through complex, low-level material. This version should be dense enough to teach internals, but smooth enough to read without feeling like a spec sheet.

1.4 L2. Under the Hood

At the lowest level, overflow and precision aren't abstract concepts—they are consequences of how CPUs, compilers, and libraries actually implement arithmetic. Understanding these details makes debugging easier and gives you control over performance, reproducibility, and correctness.

Deep Dive

1) Hardware semantics

CPUs implement integer and floating-point arithmetic directly in silicon. When the result doesn't fit, different flags or traps are triggered.

- Status flags (integers): Most architectures (x86, ARM, RISC-V) set overflow, carry, and zero flags after arithmetic. These flags drive branch instructions like `jo` ("jump if overflow").
- Floating-point control: The FPU or SIMD unit maintains exception flags (inexact, overflow, underflow, invalid, divide-by-zero). These rarely trap by default; they silently set flags until explicitly checked.

Architectural view

Arch	Integer overflow	FP behavior	Developer hooks
x86-64	Wraparound in 2's complement; OF/CF bits set	IEEE-754; flags in MXCSR	`jo`/`jno`, `fenv.h`
ARM64	Wraparound; NZCV flags	IEEE-754; exception bits	condition codes, `feset*`
RISC-V	Wraparound; OV/CF optional	IEEE-754; status regs	CSRs, trap handlers

Knowing what the CPU does lets you choose: rely on hardware wrap, trap explicitly, or add software checks.

2) Compiler and language layers

Even if hardware sets flags, your language may ignore them. Compilers often optimize based on the language spec.

- C/C++: Signed overflow is *undefined behavior*—the optimizer assumes it never happens, which can remove safety checks you thought were there.
- Rust: Catches overflow in debug builds, then forces you to pick: `checked_add`, `wrapping_add`, or `saturating_add`.
- JVM languages (Java, Kotlin, Scala): Always wrap, hiding UB but forcing you to detect overflow yourself.
- .NET (C#, F#): Defaults to wrapping; you can enable `checked` contexts to trap.
- Python: Emulates unbounded integers, but sometimes simulates C-like behavior for low-level modules.

These choices aren't arbitrary—they reflect trade-offs between speed, safety, and backward compatibility.

3) Precision management in floating point

Floating-point has more than just rounding errors. Engineers deal with gradual underflow, denormals, and fused operations.

- Subnormals: Numbers smaller than ~2.2e-308 in double precision become "denormalized," losing precision but extending the range toward zero. Many CPUs handle these slowly.
- Flush-to-zero: Some systems skip subnormals entirely, treating them as zero to boost speed. Great for graphics; risky for scientific code.
- FMA (fused multiply-add): Computes `(a*b + c)` with one rounding, often improving precision and speed. However, it can break reproducibility across machines that do/don't use FMA.

Precision events

Event	What happens	Why it matters
Overflow	Becomes ±Inf	Detectable via `isinf`, often safe
Underflow	Becomes 0 or subnormal	Performance hit, possible accuracy loss
Inexact	Result rounded	Happens constantly; only matters if flagged
Invalid	NaN produced	Division 0/0, sqrt(-1), etc.

When performance bugs show up in HPC or ML code, denormals and FMAs are often the hidden cause.

4) Debugging and testing tools

Low-level correctness requires instrumentation. Fortunately, toolchains give you options.

- Sanitizers: `-fsanitize=undefined` (Clang/GCC) traps on signed overflow.
- Valgrind / perf counters: Can catch denormal slowdowns.
- Unit-test utilities: Rust's `assert_eq!(checked_add(…))`, Python's `math.isclose`, Java's `BigDecimal` reference checks.
- Reproducibility flags: `-ffast-math` (fast but non-deterministic), vs. `-frounding-math` (strict).

Testing with multiple compilers and settings reveals assumptions you didn't know you had.

5) Strategies in production systems

When deploying real systems, you pick policies that match domain needs.

- Databases: Use `DECIMAL(p,s)` to store fixed-point, preventing float drift in sums.
- Financial systems: Explicit fixed-point types (cents as `int64`) + saturating logic on overflow.
- Graphics / ML: Accept `float32` imprecision; gain throughput with fused ops and flush-to-zero.
- Low-level kernels: Exploit modular wraparound deliberately for hash tables, checksums, and crypto.

Policy menu

Scenario	Strategy
Money transfers	Fixed-point, saturating arithmetic
Physics sim	`float64`, stable integrators, compensated summation
Hashing / RNG	Embrace wraparound modular math

Scenario	Strategy
Critical counters	`uint64` with explicit overflow trap

Thinking in policies avoids one-off hacks. Document “why” once, then apply consistently.

Code Examples

C (wrap vs check)

```c
#include <stdint.h>
#include <stdbool.h>
#include <limits.h>

bool add_checked_i32(int32_t a, int32_t b, int32_t *out) {
    if ((b > 0 && a > INT32_MAX - b) ||
        (b < 0 && a < INT32_MIN - b)) {
        return false; // overflow
    }
    *out = a + b;
    return true;
}
```

Rust (explicit intent)

```rust
fn demo() {
    let x: i32 = i32::MAX;
    println!("{:?}", x.wrapping_add(1));   // wrap
    println!("{:?}", x.checked_add(1));    // None
    println!("{:?}", x.saturating_add(1)); // clamp
}
```

Python (reproducibility check)

```python
import math

def ulp_diff(a: float, b: float) -> int:
    # Compares floats in terms of ULPs
    import struct
    ai = struct.unpack('!q', struct.pack('!d', a))[0]
    bi = struct.unpack('!q', struct.pack('!d', b))[0]
```

```
    return abs(ai - bi)

print(ulp_diff(1.0, math.nextafter(1.0, 2.0)))  # 1
```

These snippets show how different languages force you to state your policy, rather than relying on "whatever the hardware does."

Why it matters

- Performance: Understanding denormals and FMAs can save orders of magnitude in compute-heavy workloads.
- Correctness: Database money columns or counters in billing systems can silently corrupt without fixed-point or overflow checks.
- Portability: Code that relies on UB may "work" on GCC Linux but fail on Clang macOS.
- Security: Integer overflow bugs (e.g., buffer length miscalculation) remain a classic vulnerability class.

In short, overflow and precision are not "just math"—they are systems-level contracts that must be understood and enforced.

Exercises

1. Compiler behavior: Write a C function that overflows `int32_t`. Compile with and without `-fsanitize=undefined`. What changes?
2. FMA investigation: Run a dot-product with and without `-ffast-math`. Measure result differences across compilers.
3. Denormal trap: Construct a loop multiplying by `1e-308`. Time it with flush-to-zero enabled vs disabled.
4. Policy design: For an in-memory database, define rules for counters, timestamps, and currency columns. Which use wrapping, which use fixed-point, which trap?
5. Cross-language test: Implement `add_checked_i32` in C, Rust, and Python. Run edge-case inputs (`INT_MAX`, `INT_MIN`). Compare semantics.
6. ULP meter: Write a function in your language to compute ULP distance between two floats. Use it to compare rounding differences between platforms.

Chapter 2. Arrays

Chapter 3. Strings

Chapter 4. Linked Lists

Chapter 5. Stacks and Queues