The Little Book of Algorithms

Version 0.1.1

Duc-Tam Nguyen

2025-09-11

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

2.1 Static Arrays

2.2 L0 — Arrays That Grow

A dynamic array is like a container that can expand and shrink as needed. Unlike static arrays, which must know their size in advance, a dynamic array adapts as elements are added or removed. You can think of it as a bookshelf where new shelves appear automatically when space runs out. The underlying idea is simple: keep the benefits of fast index-based access, while adding flexibility to change the size.

Deep Dive

A dynamic array begins with a fixed amount of space called its capacity. When the number of elements (the length) exceeds this capacity, the array grows. This is usually done by allocating a new, larger block of memory and copying the old elements into it. After this, new elements can be added until the new capacity is filled, at which point the process repeats.

Despite this resizing process, the key properties remain:

- Fast access and update: Elements can still be reached instantly using an index.
- Append flexibility: New elements can be added at the end without worrying about fixed size.
- Occasional resizing cost: Most appends are quick, but when resizing happens, it takes longer because all elements must be copied.

The performance picture is intuitive:

Operation	Time Complexity (Typical)	Notes
Access element	O(1)	Index maps directly to position
Update element	O(1)	Replace value in place
Append element	O(1) amortized	Occasionally O(n) when resizing occurs
Pop element	O(1)	Remove from end
Insert/Delete	O(n)	Elements must be shifted

Dynamic arrays therefore trade predictability for flexibility. The occasional slow operation is outweighed by the ability to grow and shrink on demand, which makes them useful for most real-world tasks where the number of elements is not known in advance.

Worked Example

```
# Create a dynamic array using Python's built-in list
arr = []

# Append elements (array grows automatically)
for i in range(5):
    arr.append((i + 1) * 10)

print("Array after appending:", arr)

# Access and update elements
print("Element at index 2:", arr[2])
arr[2] = 99
print("Updated array:", arr)

# Remove last element
last = arr.pop()
print("Removed element:", last)
print("Array after pop:", arr)

# Traverse array
for i in range(len(arr)):
    print(f"Index {i}: {arr[i]}")
```

This short program shows how a dynamic array in Python resizes automatically with `append` and shrinks with `pop`. Access and updates remain instant, while resizing happens invisibly when more space is needed.

Why it matters

Dynamic arrays combine efficiency and flexibility. They allow programs to handle unknown or changing amounts of data without predefining sizes. They form the backbone of lists in high-level languages, balancing performance with usability. They also illustrate the idea of amortized cost: most operations are fast, but occasional expensive operations are averaged out over time.

Exercises

1. Create an array and append numbers 1 through 10. Print the final array.
2. Replace the 3rd element with a new value.
3. Remove the last two elements and print the result.
4. Write a procedure that traverses a dynamic array and computes the average of its elements.
5. Explain why appending one element might sometimes be much slower than appending another, even though both look the same in code.

2.1 L1 — Static Arrays in Practice

Static arrays are one of the simplest and most reliable ways of storing data. They are defined as collections of elements laid out in a fixed-size, contiguous block of memory. Unlike dynamic arrays, their size is determined at creation and cannot be changed later. This property makes them predictable, efficient, and easy to reason about, but also less flexible when dealing with varying amounts of data.

Deep Dive

At the heart of static arrays is their memory layout. When an array is created, the program reserves a continuous region of memory large enough to hold all its elements. Each element is stored right next to the previous one. This design allows very fast access because the position of any element can be computed directly:

```
address_of(arr[i]) = base_address + (i × element_size)
```

No searching or scanning is required, only simple arithmetic. This is why reading or writing to an element at a given index is considered O(1) — constant time regardless of the array size.

The trade-offs emerge when considering insertion or deletion. Because elements are tightly packed, inserting a new element in the middle requires shifting all the subsequent elements by one position. Deleting works the same way in reverse. These operations are therefore O(n), linear in the size of the array.

The cost summary is straightforward:

Operation	Time Complexity	Notes
Access element	O(1)	Direct index calculation
Update element	O(1)	Replace in place
Traverse	O(n)	Visit each element once
Insert/Delete	O(n)	Shifting elements required

Trade-offs.

Static arrays excel when you know the size in advance. They guarantee fast access and compact memory usage because there is no overhead for resizing or metadata. However, they lack flexibility. If the array is too small, you must allocate a larger one and copy all elements over. If it is too large, memory is wasted. This is why languages like Python provide dynamic lists by default, while static arrays are used in performance-critical or resource-constrained contexts.

Use cases.

- Buffers: Fixed-size areas for network packets or hardware input.
- Lookup tables: Precomputed constants or small ranges of values (e.g., ASCII character tables).
- Static configuration data: Tables known at compile-time, where resizing is unnecessary.

Pitfalls.

Programmers must be careful of two common issues:

1. Out-of-bounds errors: Trying to access an index outside the valid range, leading to exceptions (in safe languages) or undefined behavior (in low-level languages).
2. Sizing problems: Underestimating leads to crashes, overestimating leads to wasted memory.

Static arrays are common in many programming environments. In Python, the `array` module provides a fixed-type sequence that behaves more like a C-style array. Libraries like NumPy also provide fixed-shape arrays that offer efficient memory usage and fast computations. In C and C++, arrays are part of the language itself, and they form the foundation of higher-level containers like `std::vector`.

Worked Example

```
import array

# Create a static array of integers (type 'i' = signed int)
arr = array.array('i', [0] * 5)

# Fill the array with values
for i in range(len(arr)):
    arr[i] = (i + 1) * 10
```

```
# Access and update elements
print("Element at index 2:", arr[2])
arr[2] = 99
print("Updated element at index 2:", arr[2])

# Traverse the array
print("All elements:")
for i in range(len(arr)):
    print(f"Index {i}: {arr[i]}")

# Demonstrating the limitation: trying to insert beyond capacity
try:
    arr.insert(5, 60)  # This technically works in Python's array, but resizes internally
    print("Inserted new element:", arr)
except Exception as e:
    print("Error inserting into static array:", e)
```

This code illustrates the strengths and weaknesses of static arrays. Access and updates are immediate, and traversal is simple. But the notion of a "fixed size" means that insertion and deletion are costly or, in some languages, unsupported.

Why it matters

Static arrays are the building blocks of data structures. They teach the trade-off between speed and flexibility. They remind us that memory is finite and that how data is laid out in memory directly impacts performance. Whether writing Python code, using NumPy, or implementing algorithms in C, understanding static arrays makes it easier to reason about cost, predict behavior, and avoid common errors.

Exercises

1. Create an array of size 8 and fill it with even numbers from 2 to 16. Then access the 4th element directly.
2. Update the middle element of a fixed-size array with a new value.
3. Write a procedure to traverse an array and find the maximum element.
4. Explain why inserting a new value into the beginning of a static array requires shifting every other element.
5. Give two examples of real-world systems where fixed-size arrays are a natural fit.

2.1 L2 — Static Arrays and the System Beneath

Static arrays are more than just a collection of values; they are a direct window into how computers store and access data. At the advanced level, understanding static arrays means looking at memory models, cache behavior, compiler optimizations, and the role of arrays in operating systems and production libraries. This perspective is critical for building high-performance software and for avoiding subtle, system-level bugs.

Deep Dive

At the lowest level, a static array is a contiguous block of memory. When an array is declared, the compiler calculates the required size as `length × element_size` and reserves that many bytes. Each element is addressed by simple arithmetic:

```
address_of(arr[i]) = base_address + (i × element_size)
```

This is why access and updates are constant time. The difference between static arrays and dynamically allocated ones often comes down to where the memory lives. Arrays declared inside a function may live on the stack, offering fast allocation and automatic cleanup. Larger arrays or arrays whose size isn't known at compile time are allocated on the heap, requiring runtime management via calls such as `malloc` and `free`.

The cache hierarchy makes arrays especially efficient. Because elements are contiguous, accessing `arr[i]` loads not just one element but also its neighbors into a cache line (often 64 bytes). This property, known as spatial locality, means that scanning through an array is very fast. Prefetchers in modern CPUs exploit this by pulling in upcoming cache lines before they are needed. However, irregular access patterns (e.g., striding by 17) can defeat prefetching and lead to performance drops.

Alignment and padding further influence performance. On most systems, integers must start at addresses divisible by 4, and doubles at addresses divisible by 8. If the compiler cannot guarantee alignment, it may add padding bytes to enforce it. Misaligned accesses can cause slowdowns or even hardware faults on strict architectures.

Different programming languages expose these behaviors differently. In C, a declaration like `int arr[10];` on the stack creates exactly 40 bytes on a 32-bit system. In contrast, `malloc(10 * sizeof(int))` allocates memory on the heap. In C++, `std::array<int, 10>` is a safer wrapper around C arrays, while `std::vector<int>` adds resizing at the cost of indirection and metadata. In Fortran and NumPy, multidimensional arrays can be stored in column-major order rather than row-major, which changes how indices map to addresses and affects iteration performance.

The operating system kernel makes heavy use of static arrays. For example, Linux defines fixed-size arrays in structures like `task_struct` for file descriptors, and uses arrays in page tables for managing memory mappings. Static arrays provide predictability and remove the need for runtime memory allocation in performance-critical or security-sensitive code.

From a performance profiling standpoint, arrays reveal fundamental trade-offs. Shifting elements during insertion or deletion requires copying bytes across memory, and the cost grows linearly with the number of elements. Compilers attempt to optimize loops over arrays with vectorization, turning element-wise operations into SIMD instructions. They may also apply loop unrolling or bounds-check elimination (BCE) when it can be proven that indices remain safe.

Static arrays also carry risks. In C and C++, accessing out-of-bounds memory leads to undefined behavior, often exploited in buffer overflow attacks. Languages like Java or Python mitigate this with runtime bounds checks, but at the expense of some performance.

At this level, static arrays should be seen not only as a data structure but as a fundamental contract between code, compiler, and hardware.

Worked Example (C)

```c
#include <stdio.h>

int main() {
    // Static array of 8 integers allocated on the stack
    int arr[8];

    // Initialize array
    for (int i = 0; i < 8; i++) {
        arr[i] = (i + 1) * 10;
    }

    // Access and update element
    printf("Element at index 3: %d\n", arr[3]);
    arr[3] = 99;
    printf("Updated element at index 3: %d\n", arr[3]);

    // Traverse with cache-friendly pattern
    int sum = 0;
    for (int i = 0; i < 8; i++) {
        sum += arr[i];
    }
    printf("Sum of array: %d\n", sum);
```

```
    // Dangerous: Uncommenting would cause undefined behavior
    // printf("%d\n", arr[10]);

    return 0;
}
```

This C program demonstrates how static arrays live on the stack, how indexing works, and why out-of-bounds access is dangerous. On real hardware, iterating sequentially benefits from spatial locality, making the traversal very fast compared to random access.

Why it matters

Static arrays are the substrate upon which much of computing is built. They are simple in abstraction but complex in practice, touching compilers, operating systems, and hardware. Understanding them is essential for:

- Writing cache-friendly and high-performance code.
- Avoiding security vulnerabilities like buffer overflows.
- Appreciating why higher-level data structures behave the way they do.
- Building intuition for memory layout, alignment, and the interaction between code and the CPU.

Arrays are not just "collections of values" — they are the foundation of efficient data processing.

Exercises

1. In C, declare a static array of size 16 and measure how long it takes to sum its elements sequentially versus accessing them in steps of 4. Explain the performance difference.
2. Explain why iterating over a 2D array row by row is faster in C than column by column.
3. Consider a struct with mixed types (e.g., `char`, `int`, `double`). Predict where padding bytes will be inserted if placed inside an array.
4. Research and describe how the Linux kernel uses static arrays in managing processes or memory.
5. Demonstrate with code how accessing beyond the end of a static array in C can cause undefined behavior, and explain why this is a serious risk in system programming.

2.2 Dynamic Arrays

2.2 L0 — Arrays That Grow

A dynamic array is like a container that can expand and shrink as needed. Unlike static arrays, which must know their size in advance, a dynamic array adapts as elements are added or removed. You can think of it as a bookshelf where new shelves appear automatically when space runs out. The underlying idea is simple: keep the benefits of fast index-based access, while adding flexibility to change the size.

Deep Dive

A dynamic array begins with a fixed amount of space called its capacity. When the number of elements (the length) exceeds this capacity, the array grows. This is usually done by allocating a new, larger block of memory and copying the old elements into it. After this, new elements can be added until the new capacity is filled, at which point the process repeats.

Despite this resizing process, the key properties remain:

- Fast access and update: Elements can still be reached instantly using an index.
- Append flexibility: New elements can be added at the end without worrying about fixed size.
- Occasional resizing cost: Most appends are quick, but when resizing happens, it takes longer because all elements must be copied.

The performance picture is intuitive:

Operation	Time Complexity (Typical)	Notes
Access element	O(1)	Index maps directly to position
Update element	O(1)	Replace value in place
Append element	O(1) amortized	Occasionally O(n) when resizing occurs
Pop element	O(1)	Remove from end
Insert/Delete	O(n)	Elements must be shifted

Dynamic arrays therefore trade predictability for flexibility. The occasional slow operation is outweighed by the ability to grow and shrink on demand, which makes them useful for most real-world tasks where the number of elements is not known in advance.

Worked Example

```
# Create a dynamic array using Python's built-in list
arr = []

# Append elements (array grows automatically)
for i in range(5):
    arr.append((i + 1) * 10)

print("Array after appending:", arr)

# Access and update elements
print("Element at index 2:", arr[2])
arr[2] = 99
print("Updated array:", arr)

# Remove last element
last = arr.pop()
print("Removed element:", last)
print("Array after pop:", arr)

# Traverse array
for i in range(len(arr)):
    print(f"Index {i}: {arr[i]}")
```

This short program shows how a dynamic array in Python resizes automatically with `append` and shrinks with `pop`. Access and updates remain instant, while resizing happens invisibly when more space is needed.

Why it matters

Dynamic arrays combine efficiency and flexibility. They allow programs to handle unknown or changing amounts of data without predefining sizes. They form the backbone of lists in high-level languages, balancing performance with usability. They also illustrate the idea of amortized cost: most operations are fast, but occasional expensive operations are averaged out over time.

Exercises

1. Create an array and append numbers 1 through 10. Print the final array.

2. Replace the 3rd element with a new value.
3. Remove the last two elements and print the result.
4. Write a procedure that traverses a dynamic array and computes the average of its elements.
5. Explain why appending one element might sometimes be much slower than appending another, even though both look the same in code.

2.2 L1 — Dynamic Arrays in Practice

Dynamic arrays extend the idea of static arrays by making size flexible. They allow adding or removing elements without knowing the total number in advance. Under the hood, this flexibility is achieved through careful memory management: the array is stored in a contiguous block, but when more space is needed, a larger block is allocated, and all elements are copied over. This mechanism balances speed with adaptability and is the reason why dynamic arrays are the default sequence type in many languages.

Deep Dive

A dynamic array starts with a certain capacity, often larger than the initial number of elements. When the number of stored elements exceeds capacity, the array is resized. The common strategy is to double the capacity. For example, an array of capacity 4 that becomes full will reallocate to capacity 8. All existing elements are copied into the new block, and the old memory is freed.

This strategy makes appending efficient on average. While an individual resize costs O(n) because of the copying, most appends are O(1). Across a long sequence of operations, the total cost averages out — this is called amortized analysis.

Dynamic arrays retain the key advantages of static arrays:

- Contiguous storage means fast random access with `O(1)` time.
- Updates are also `O(1)` because they overwrite existing slots.

The challenges appear with other operations:

- Insertions or deletions in the middle require shifting elements, making them O(n).
- Resizing events create temporary latency spikes, especially when arrays are large.

A clear summary:

Operation	Time Complexity	Notes
Access element	O(1)	Direct index calculation
Update element	O(1)	Replace value in place
Append element	O(1) amortized	Occasional O(n) when resizing

Operation	Time Complexity	Notes
Pop element	O(1)	Remove from end
Insert/Delete	O(n)	Shifting elements required

Trade-offs.

Dynamic arrays sacrifice predictability for convenience. Resizing causes performance spikes, but the doubling strategy keeps the average cost low. Over-allocation wastes some memory, but it reduces the frequency of resizes. The key is that this trade-off is usually favorable in practice.

Use cases.

Dynamic arrays are well-suited for:

- Lists whose size is not known in advance.
- Workloads dominated by appending and reading values.
- General-purpose data structures in high-level programming languages.

Language implementations.

- Python: `list` is a dynamic array, using an over-allocation strategy to reduce frequent resizes.
- C++: `std::vector` doubles its capacity when needed, invalidating pointers/references after reallocation.
- Java: `ArrayList` grows by about 1.5× when full, trading memory efficiency for fewer copies.

Pitfalls.

- In languages with pointers or references, resizes can invalidate existing references.
- Large arrays may cause noticeable latency during reallocation.
- Middle insertions and deletions remain inefficient compared to linked structures.

Worked Example

```python
# Demonstrate dynamic array behavior using Python's list
arr = []

# Append elements to trigger resizing internally
for i in range(12):
    arr.append(i)
    print(f"Appended {i}, length = {len(arr)}")

# Access and update
print("Element at index 5:", arr[5])
arr[5] = 99
print("Updated element at index 5:", arr[5])

# Insert in the middle (expensive operation)
arr.insert(6, 123)
print("Array after middle insert:", arr)

# Pop elements
arr.pop()
print("Array after pop:", arr)
```

This example illustrates appending, updating, inserting, and popping. While Python hides the resizing, the cost is there: occasionally the list must allocate more space and copy its contents.

Why it matters

Dynamic arrays balance flexibility and performance. They demonstrate the principle of amortized complexity, showing how expensive operations can be smoothed out over time. They also highlight trade-offs between memory usage and speed. Understanding them explains why high-level lists perform well in everyday coding but also where they can fail under stress.

Exercises

1. Create a dynamic array and append the numbers 1 to 20. Measure how many times resizing would have occurred if the growth factor were 2.
2. Insert an element into the middle of a large array and explain why this operation is slower than appending at the end.
3. Write a procedure to remove all odd numbers from a dynamic array.
4. Compare Python's `list`, Java's `ArrayList`, and C++'s `std::vector` in terms of growth strategy.

5. Explain why references to elements of a `std::vector` may become invalid after resizing.

2.2 L2 — Dynamic Arrays Under the Hood

Dynamic arrays reveal how high-level flexibility is built on top of low-level memory management. While they appear as resizable containers, underneath they are carefully engineered to balance performance, memory efficiency, and safety. Understanding their internals sheds light on allocators, cache behavior, and the risks of pointer invalidation.

Deep Dive

Dynamic arrays rely on heap allocation. When first created, they reserve a contiguous memory block with some capacity. As elements are appended and the array fills, the implementation must allocate a new, larger block, copy all existing elements, and free the old block.

Most implementations use a geometric growth strategy, often doubling the capacity when space runs out. Some use a factor smaller than two, such as 1.5×, to reduce memory waste. The trade-off is between speed and efficiency:

- Larger growth factors reduce the number of costly reallocations.
- Smaller growth factors waste less memory but increase resize frequency.

This leads to an amortized O(1) cost for append. Each resize is expensive, but they happen infrequently enough that the average cost remains constant across many operations.

However, resizes have side effects:

- Pointer invalidation: In C++ `std::vector`, any reference, pointer, or iterator into the old memory becomes invalid after reallocation.
- Latency spikes: Copying thousands or millions of elements in one step can stall a program, especially in real-time or low-latency systems.
- Allocator fragmentation: Repeated growth and shrink cycles can fragment the heap, reducing performance in long-running systems.

Cache efficiency is one of the strengths of dynamic arrays. Because elements are stored contiguously, traversals are cache-friendly, and prefetchers can load entire blocks into cache lines. But reallocations can disrupt locality temporarily, as the array may move to a new region of memory.

Different languages implement dynamic arrays with variations:

- Python lists use over-allocation with a small growth factor (~12.5% to 25% extra). This minimizes wasted memory while keeping amortized costs stable.

- C++ `std::vector` typically doubles its capacity when needed. Developers can call `reserve()` to preallocate memory and avoid repeated reallocations.
- Java `ArrayList` grows by ~1.5×, balancing heap usage with resize frequency.

Dynamic arrays also face risks:

- If resizing logic is incorrect, buffer overflows may occur.
- Attackers can exploit repeated growth/shrink cycles to cause denial-of-service via frequent allocations.
- Very large allocations can fail outright if memory is exhausted.

From a profiling perspective, workloads matter. Append-heavy patterns perform extremely well due to amortization. Insert-heavy or middle-delete workloads perform poorly because of element shifting. Allocator-aware optimizations, like pre-reserving capacity, can dramatically improve performance.

Worked Example (C++)

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v;
    v.reserve(4);  // reserve space for 4 elements to reduce reallocations

    for (int i = 0; i < 12; i++) {
        v.push_back(i * 10);
        std::cout << "Appended " << i*10
                  << ", size = " << v.size()
                  << ", capacity = " << v.capacity() << std::endl;
    }

    // Access and update
    std::cout << "Element at index 5: " << v[5] << std::endl;
    v[5] = 99;
    std::cout << "Updated element at index 5: " << v[5] << std::endl;

    // Demonstrate invalidation risk
    int* ptr = &v[0];
    v.push_back(12345); // may reallocate and move data
    std::cout << "Old pointer may now be invalid: " << *ptr << std::endl; // UB if reallocat
}
```

This program shows how `std::vector` manages capacity. The output reveals how capacity grows as more elements are appended. The pointer invalidation example highlights a subtle but critical risk: after a resize, old addresses into the array are no longer safe.

Why it matters

Dynamic arrays expose the tension between abstraction and reality. They appear simple, but internally they touch almost every layer of the system: heap allocators, caches, compiler optimizations, and safety checks. They are essential for understanding how high-level languages achieve both usability and performance, and they illustrate real-world engineering trade-offs between speed, memory, and safety.

Exercises

1. In C++, measure the capacity growth of a `std::vector<int>` as you append 1,000 elements. Plot size vs capacity.
2. Explain why a program that repeatedly appends and deletes elements might fragment the heap over time.
3. Compare the growth strategies of Python `list`, C++ `std::vector`, and Java `ArrayList`. Which wastes more memory? Which minimizes resize cost?
4. Write a program that appends 1 million integers to a dynamic array and then times the traversal. Compare it with inserting 1 million integers at the beginning.
5. Show how `reserve()` in `std::vector` or `ensureCapacity()` in Java `ArrayList` can eliminate costly reallocation spikes.

2.3 Slices & Views

2.3 L0 — Looking Through a Window

A slice or view is a way to look at part of an array without creating a new one. Instead of copying data, a slice points to the same underlying elements, just with its own start and end boundaries. This makes working with subarrays fast and memory-efficient. You can think of a slice as a window into a longer row of boxes, showing only the portion you care about.

Deep Dive

When you take a slice, you don't get a new array filled with copied elements. Instead, you get a new "view" that remembers where in the original array it starts and stops. This is useful because:

- No copying means creating a slice is very fast.
- Shared storage means changes in the slice also affect the original array (in languages like Go, Rust, or NumPy).
- Reduced scope means you can focus on a part of the array without carrying the entire structure.

Key properties of slices:

1. They refer to the same memory as the original array.
2. They have their own length (number of elements visible).
3. They may also carry a capacity, which limits how far they can expand into the original array.

In Python, list slicing (`arr[2:5]`) creates a new list with copies of the elements. This is not a true view. By contrast, NumPy arrays, Go slices, and Rust slices provide real views — updates to the slice affect the original array.

A summary:

Feature	Slice/View	New Array (Copy)
Memory usage	Shares existing storage	Allocates new storage
Creation cost	O(1)	O(n) for copied elements
Updates	Affect original array	Independent
Safety	Risk of aliasing issues	No shared changes

Slices are especially valuable when working with large datasets, where copying would be too expensive.

Worked Example

```
# Python slicing creates a copy, but useful to illustrate concept
arr = [10, 20, 30, 40, 50]

# Slice of middle part
sub = arr[1:4]
print("Original array:", arr)
print("Slice (copy in Python):", sub)

# Modifying the slice does not affect the original (Python behavior)
sub[0] = 99
print("Modified slice:", sub)
```

```
print("Original array unchanged:", arr)

# In contrast, NumPy arrays behave like true views
import numpy as np
arr_np = np.array([10, 20, 30, 40, 50])
sub_np = arr_np[1:4]
sub_np[0] = 99
print("NumPy slice reflects back:", arr_np)
```

This example shows the difference: Python lists create a copy, while NumPy slices act as views and affect the original.

Why it matters

Slices let you work with subsets of data without wasting memory or time copying. They are critical in systems and scientific computing where performance matters. They also highlight the idea of aliasing: when two names refer to the same data. Understanding slices teaches you when changes propagate and when they don't, which helps avoid surprising bugs.

Exercises

1. Create an array of 10 numbers. Take a slice of the middle 5 elements and print them.
2. Update the first element in your slice and describe what happens to the original array in your chosen language.
3. Compare slicing behavior in Python and NumPy: which one copies, which one shares?
4. Explain why slicing a very large dataset is more efficient than copying it.
5. Think of a real-world analogy where two people share the same resource but only see part of it. How does this relate to slices?

2.3 L1 — Slices in Practice

Slices provide a practical way to work with subarrays efficiently. Instead of copying data into a new structure, a slice acts as a lightweight reference to part of an existing array. This gives programmers flexibility to manipulate sections of data without paying the cost of duplication, while still preserving the familiar indexing model of arrays.

Deep Dive

At the implementation level, a slice is typically represented by a small structure that stores:

1. A pointer to the first element in the slice.
2. The slice's length (how many elements it can access).
3. Optionally, its capacity (how far the slice can grow into the backing array).

Indexing into a slice works just like indexing into an array:

```
slice[i] → base_address + i × element_size
```

The complexity model stays consistent:

- Slice creation: O(1) when implemented as a view, O(n) if the language copies elements.
- Access/update: O(1), just like arrays.
- Traversal: O(k), proportional to the slice's length.

This design makes slices efficient but introduces trade-offs. With true views, the slice and the original array share memory. Updates made through one are visible through the other. This can be extremely useful but also dangerous, as it introduces the possibility of unintended side effects. Languages that prioritize safety (like Python lists) avoid this by returning a copy instead of a view.

The balance is clear:

- Views (Go, Rust, NumPy): fast and memory-efficient, but require discipline to avoid aliasing bugs.
- Copies (Python lists): safer, but slower and more memory-intensive for large arrays.

A summary of behaviors:

Language/Library	Slice Behavior	Shared Updates	Notes
Go	View	Yes	Backed by `(ptr, len, cap)` triple
Rust	View	Yes	Safe with borrow checker (mutable/immutable)
Python list	Copy	No	Safer but memory-expensive
NumPy array	View	Yes	Basis of efficient scientific computing
C/C++	Manual pointer	Yes	No built-in slice type; must manage manually

Use cases.

- Processing large datasets in segments without copying.
- Implementing algorithms like sliding windows, partitions, or block-based iteration.
- Sharing views of arrays across functions for modular design without allocating new memory.

Pitfalls.

- In languages with views, careless updates can corrupt the original array unexpectedly.
- In Go and C++, extending a slice/view beyond its capacity causes runtime errors or undefined behavior.
- In Python, forgetting that slices are copies can lead to performance issues in large-scale workloads.

Worked Example

```
# Demonstrating copy slices vs view slices in Python and NumPy

# Python list slicing creates a copy
arr = [1, 2, 3, 4, 5]
sub = arr[1:4]
sub[0] = 99
print("Python original:", arr)  # unchanged
print("Python slice (copy):", sub)

# NumPy slicing creates a view
import numpy as np
arr_np = np.array([1, 2, 3, 4, 5])
sub_np = arr_np[1:4]
sub_np[0] = 99
print("NumPy original (affected):", arr_np)
print("NumPy slice (view):", sub_np)
```

This example shows the key difference: Python lists copy, while NumPy provides true views. The choice reflects different design priorities: safety in Python's core data structures versus performance in numerical computing.

Why it matters

Slices make programs more efficient and expressive. They eliminate unnecessary copying, speed up algorithms that work on subranges, and support modular programming by passing references instead of duplicating data. At the same time, they expose important design trade-offs between safety and performance. Understanding slices provides insight into how modern languages manage memory efficiently while protecting against common errors.

Exercises

1. In Go, create an array of 10 elements and take a slice of the middle 5. Update the slice and observe the effect on the array.
2. In Python, slice a list of 1 million numbers and explain the performance cost compared to slicing a NumPy array of the same size.
3. Write a procedure that accepts a slice and doubles each element. Test with both a copy-based language (Python lists) and a view-based language (NumPy or Go).
4. Explain why passing slices to functions is more memory-efficient than passing entire arrays.
5. Discuss a scenario where slice aliasing could lead to unintended bugs in a large program.

2.3 L2 — Slices and Views in Systems

Slices are not just convenient programming shortcuts; they represent a powerful abstraction that ties language semantics to hardware realities. At this level, slices expose details about memory layout, lifetime, and compiler optimizations. They are central to performance-critical systems because they allow efficient access to subsets of data without copying, while also demanding careful handling to avoid aliasing bugs and unsafe memory access.

Deep Dive

A slice is typically represented internally as a triple:

- A pointer to the first element,
- A length describing how many elements are visible,
- A capacity showing how far the slice may extend into the backing array.

Indexing into a slice is still O(1), but the compiler inserts bounds checks to prevent invalid access. In performance-sensitive code, compilers often apply bounds-check elimination (BCE) when they can prove that loop indices remain within safe limits. This allows slices to combine safety with near-native performance.

Slices are non-owning references. They do not manage memory themselves but instead depend on the underlying array. In languages like Rust, the borrow checker enforces lifetimes to prevent dangling slices. In C and C++, however, programmers must manually ensure that the backing array outlives the slice, or risk undefined behavior.

Because slices share memory, they introduce aliasing. Multiple slices can point to overlapping regions of the same array. This can lead to subtle bugs if two parts of a program update the same region concurrently. In multithreaded contexts, mutable aliasing without synchronization can cause data races. Some systems adopt copy-on-write strategies to reduce risks, but this adds overhead.

From a performance perspective, slices preserve contiguity, which is ideal for cache locality and prefetching. Sequential traversal is cache-friendly, but strided access (e.g., every 3rd element) can defeat hardware prefetchers, reducing efficiency. Languages like NumPy exploit strides explicitly, enabling both dense and sparse-like views without copying.

Language designs differ in how they handle slices:

- Go uses `(ptr, len, cap)`. Appending to a slice may allocate a new array if capacity is exceeded, silently detaching it from the original backing storage.
- Rust distinguishes `&[T]` for immutable and `&mut [T]` for mutable slices, with the compiler enforcing safe borrowing rules.
- C/C++ provide no built-in slice type, so developers rely on raw pointers and manual length tracking. This is flexible but error-prone.
- NumPy supports advanced slicing: views with strides, broadcasting rules, and multidimensional slices for scientific computing.

Compilers also optimize slice-heavy code:

- Vectorization transforms element-wise loops into SIMD instructions when slices are contiguous.
- Escape analysis determines whether slices can stay stack-allocated or must be promoted to the heap.

System-level use cases highlight the importance of slices:

- Zero-copy I/O: network and file system buffers are exposed as slices into larger memory regions.
- Memory-mapped files: slices map directly to disk pages, enabling efficient processing of large datasets.
- GPU programming: CUDA and OpenCL kernels operate on slices of device memory, avoiding transfers.

These applications show why slices are not just a programming convenience but a core tool for bridging high-level logic with low-level performance.

Worked Example (Go)

```go
package main

import "fmt"

func main() {
    arr := [6]int{10, 20, 30, 40, 50, 60}
    s := arr[1:4] // slice referencing elements 20, 30, 40

    fmt.Println("Original array:", arr)
    fmt.Println("Slice view:", s)

    // Update through slice
    s[0] = 99
    fmt.Println("After update via slice, array:", arr)

    // Demonstrate capacity
    fmt.Println("Slice length:", len(s), "capacity:", cap(s))

    // Appending beyond slice capacity reallocates
    s = append(s, 70, 80)
    fmt.Println("Slice after append:", s)
    fmt.Println("Array after append (unchanged):", arr)
}
```

This example illustrates Go's slice model. The slice `s` initially shares storage with `arr`. Updates propagate to the array. However, when appending exceeds the slice's capacity, Go allocates a new backing array, breaking the link with the original. This behavior is efficient but can surprise developers if not understood.

Why it matters

Slices embody key system concepts: pointer arithmetic, memory ownership, cache locality, and aliasing. They explain how languages achieve zero-copy abstractions while balancing safety and performance. They also highlight risks such as dangling references and silent reallocations. Mastery of slices is essential for building efficient algorithms, avoiding memory errors, and reasoning about system-level performance.

Exercises

1. In Go, create an array of 8 integers and take overlapping slices. Modify one slice and observe effects on the other. Explain why this happens.
2. In Rust, attempt to create two mutable slices of the same array region. Explain why the borrow checker rejects it.
3. In C, simulate a slice using a pointer and a length. Show what happens if the backing array is freed while the slice is still in use.
4. In NumPy, create a 2D array and take a strided slice (every second row). Explain why performance is worse than contiguous slicing.
5. Compare how Python, Go, and Rust enforce (or fail to enforce) safety when working with slices.

2.4 Multidimensional Arrays

2.4 L0 — Tables and Grids

A multidimensional array is an extension of the simple array idea. Instead of storing data in a single row, a multidimensional array organizes elements in a grid, table, or cube. The most common example is a two-dimensional array, which looks like a table with rows and columns. Each position in the grid is identified by two coordinates: one for the row and one for the column. This structure is useful for representing spreadsheets, images, game boards, and mathematical matrices.

Deep Dive

You can think of a multidimensional array as an array of arrays. A two-dimensional array is a list where each element is itself another list. For example, a 3×3 table contains 3 rows, each of which has 3 columns. Accessing an element requires specifying both coordinates: `arr[row][col]`.

Even though we visualize multidimensional arrays as grids, in memory they are still stored as a single continuous sequence. To find an element, the program computes its position using a formula. In a 2D array with `n` columns, the element at `(row, col)` is located at:

```
index = row × n + col
```

This mapping allows direct access in constant time, just like with 1D arrays.

Common operations are:

- Creation: decide dimensions and initialize with values.
- Access: specify row and column to retrieve an element.
- Update: change the value at a given coordinate.
- Traversal: visit elements row by row or column by column.

A quick summary:

Operation	Description	Cost
Access element	Get value at (row, col)	O(1)
Update element	Replace value at (row, col)	O(1)
Traverse array	Visit all elements	O(n×m)

Multidimensional arrays introduce an important detail: traversal order. In many languages (like C and Python's NumPy), arrays are stored in row-major order, which means all elements of the first row are laid out contiguously, then the second row, and so on. Others, like Fortran, use column-major order. This difference affects performance in more advanced topics, but at this level, the key idea is that access is still fast and predictable.

Worked Example

```
# Create a 2D array (3x3 table) using list of lists
table = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Access element in second row, third column
print("Element at (1, 2):", table[1][2])  # prints 6

# Update element
table[0][0] = 99
print("Updated table:", table)

# Traverse row by row
print("Row traversal:")
for row in table:
    for val in row:
        print(val, end=" ")
    print()
```

This example shows how to build and use a 2D array in Python. It looks like a table, with easy access via coordinates.

Why it matters

Multidimensional arrays provide a natural way to represent structured data like matrices, grids, and images. They allow algorithms to work directly with two-dimensional or higher-dimensional information without flattening everything into one long row. This makes programs easier to write, read, and reason about.

Exercises

1. Create a 3×3 array with numbers 1 through 9 and print it in a table format.
2. Access the element at row 2, column 3 and describe how you found it.
3. Change the center element of a 3×3 array to 0.
4. Write a loop to compute the sum of all values in a 4×4 array.
5. Explain why accessing `(row, col)` in a 2D array is still O(1) even though the data is stored in a single sequence in memory.

2.4 L1 — Multidimensional Arrays in Practice

Multidimensional arrays are powerful because they extend the linear model of arrays into grids, tables, and higher dimensions. At a practical level, they are still stored in memory as a flattened linear block. What changes is the indexing formula: instead of a single index, we use multiple coordinates that the system translates into one offset.

Deep Dive

The most common form is a 2D array. In memory, the elements are laid out row by row (row-major) or column by column (column-major).

- Row-major (C, NumPy default): elements of each row are contiguous.
- Column-major (Fortran, MATLAB): elements of each column are contiguous.

For a 2D array with `num_cols` columns, the element at `(row, col)` in row-major order is located at:

```
index = row × num_cols + col
```

For column-major order with `num_rows` rows, the formula is:

```
index = col × num_rows + row
```

This distinction matters when traversing. Accessing elements in the memory's natural order (row by row for row-major, column by column for column-major) is cache-friendly. Traversing in the opposite order forces the program to jump around in memory, leading to slower performance.

Extending to 3D and higher is straightforward. For a 3D array with `(layers, rows, cols)` in row-major order:

```
index = layer × (rows × cols) + row × cols + col
```

Complexity remains consistent:

- Access/update: O(1) using index calculation.
- Traversal: O(n × m) for 2D, O(n × m × k) for 3D.

Trade-offs:

- Contiguous multidimensional arrays provide excellent performance for predictable workloads (e.g., matrix operations).
- Resizing is costly because the entire block must be reallocated.
- Jagged arrays (arrays of arrays) provide flexibility but lose memory contiguity, reducing cache performance.

Use cases:

- Storing images (pixels as grids).
- Mathematical matrices in scientific computing.
- Game boards and maps.
- Tables in database-like structures.

Different languages implement multidimensional arrays differently:

- Python lists: nested lists simulate 2D arrays but are jagged and fragmented in memory.
- NumPy: provides true multidimensional arrays stored contiguously in row-major (default) or column-major order.
- C/C++: support both contiguous multidimensional arrays (`int arr[rows][cols];`) and pointer-based arrays of arrays.
- Java: uses arrays of arrays (jagged by default).

Worked Example

```python
# Comparing list of lists vs NumPy arrays
# List of lists (jagged)
table = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("Element at (2, 1):", table[2][1])  # 8

# NumPy array (true contiguous 2D array)
import numpy as np
matrix = np.array([[1,2,3],[4,5,6],[7,8,9]])
print("Element at (2, 1):", matrix[2,1])  # 8

# Traversal in row-major order
for row in range(matrix.shape[0]):
    for col in range(matrix.shape[1]):
        val = matrix[row, col]  # efficient in NumPy
```

The Python list-of-lists behaves like a table, but each row may live separately in memory. NumPy, on the other hand, stores data contiguously, enabling much faster iteration and vectorized operations.

Why it matters

Multidimensional arrays are central to real-world applications, from graphics and simulations to data science and machine learning. They highlight how physical memory layout (row-major vs column-major) interacts with algorithm design. Understanding them allows developers to choose between safety, flexibility, and performance, depending on the problem.

Exercises

1. Write a procedure to sum all values in a 5×5 array by traversing row by row.
2. For a 3×3 NumPy array, access element `(2,1)` and explain how its memory index is calculated in row-major order.
3. Create a jagged array (rows of different lengths) in Python. Show how traversal differs from a true 2D array.
4. Explain why traversing a NumPy array by rows is faster than by columns.
5. Write a formula for computing the linear index of `(i,j,k)` in a 3D array stored in row-major order.

2.4 L2 — Multidimensional Arrays and System Realities

Multidimensional arrays are not only a logical abstraction but also a system-level structure that interacts with memory layout, caches, and compilers. At this level, understanding how they are stored, accessed, and optimized is essential for building high-performance code in scientific computing, graphics, and data-intensive systems.

Deep Dive

A multidimensional array is stored either as a contiguous linear block or as an array of pointers (jagged array). In the contiguous layout, elements follow one another in memory according to a linearization formula. In row-major order (C, NumPy), a 2D element at `(row, col)` is:

```
index = row × num_cols + col
```

In column-major order (Fortran, MATLAB), the formula is:

```
index = col × num_rows + row
```

This difference has deep performance consequences. In row-major layout, traversing row by row is cache-friendly because consecutive elements are contiguous. Traversing column by column introduces large strides, which can cause cache and TLB misses. In column-major arrays, the reverse holds true.

Cache and performance.

When an array is traversed sequentially in its natural memory order, cache lines are used efficiently and hardware prefetchers work well. Strided access, such as reading every k-th column in a row-major layout, prevents prefetchers from predicting the access pattern and leads to performance drops. For large arrays, this can mean the difference between processing gigabytes per second and megabytes per second.

Alignment and padding.

Compilers and libraries often align rows to cache line or SIMD vector boundaries. For example, a 64-byte cache line may cause padding to be inserted so that each row begins on a boundary. In parallel systems, this prevents false sharing when multiple threads process different rows. However, padding increases memory footprint.

Language-level differences.

- C/C++: contiguous 2D arrays (`int arr[rows][cols]`) guarantee row-major layout. Jagged arrays (array of pointers) sacrifice locality but allow uneven row sizes.
- Fortran/MATLAB: column-major ordering dominates scientific computing, influencing algorithms in BLAS and LAPACK.
- NumPy: stores strides explicitly, enabling flexible slicing and arbitrary views. Strided slices can represent transposed matrices without copying.

Optimizations.

- Loop tiling/blocking: partition loops into smaller blocks that fit into cache, maximizing reuse.
- SIMD-friendly layouts: structure-of-arrays (SoA) improves vectorization compared to array-of-structures (AoS).
- Matrix multiplication kernels: carefully designed to exploit cache hierarchy, prefetching, and SIMD registers.

System-level use cases.

- Image processing: images stored as row-major arrays, with pixels in contiguous scanlines. Efficient filters process them row by row.
- GPU computing: memory coalescing requires threads in a warp to access contiguous memory regions; array layout directly affects throughput.
- Databases: columnar storage uses column-major arrays, enabling fast scans and aggregation queries.

Pitfalls.

- Traversing in the "wrong" order can cause performance cliffs.
- Large index calculations may overflow if not handled carefully.
- Porting algorithms between row-major and column-major languages can introduce subtle bugs.

Profiling. Practical analysis involves comparing traversal patterns, cache miss rates, and vectorization efficiency. Modern compilers can eliminate redundant bounds checks and auto-vectorize well-structured loops, but poor layout or order can block these optimizations.

Worked Example (C)

```c
#include <stdio.h>
#define ROWS 4
#define COLS 4

int main() {
    int arr[ROWS][COLS];

    // Fill the array
    for (int r = 0; r < ROWS; r++) {
        for (int c = 0; c < COLS; c++) {
            arr[r][c] = r * COLS + c;
        }
    }

    // Row-major traversal (cache-friendly in C)
    int sum_row = 0;
    for (int r = 0; r < ROWS; r++) {
        for (int c = 0; c < COLS; c++) {
            sum_row += arr[r][c];
        }
    }

    // Column traversal (less efficient in row-major)
    int sum_col = 0;
    for (int c = 0; c < COLS; c++) {
        for (int r = 0; r < ROWS; r++) {
            sum_col += arr[r][c];
        }
    }

    printf("Row traversal sum: %d\n", sum_row);
    printf("Column traversal sum: %d\n", sum_col);
    return 0;
}
```

This program highlights traversal order. On large arrays, row-major traversal is much faster in C because of cache-friendly memory access, while column traversal may cause frequent cache misses.

Why it matters

Multidimensional arrays sit at the heart of performance-critical applications. Their memory layout determines how well algorithms interact with CPU caches, vector units, and GPUs. Understanding row-major vs column-major, stride penalties, and cache-aware traversal allows developers to write software that scales from toy programs to high-performance computing systems.

Exercises

1. In C, create a 1000×1000 matrix and measure the time difference between row-major and column traversal. Explain the results.
2. In NumPy, take a 2D array and transpose it. Use `.strides` to confirm that the transposed array is a view, not a copy.
3. Write the linear index formula for a 4D array `(a,b,c,d)` in row-major order.
4. Explain how false sharing could occur when two threads update adjacent rows of a large array.
5. Compare the impact of row-major vs column-major layout in matrix multiplication performance.

2.5 Sparse Arrays

2.5 L0 — Sparse Arrays as Empty Parking Lots

A sparse array is a way of storing data when most of the positions are empty. Instead of recording every slot like in a dense array, a sparse array only remembers the places that hold actual values. You can think of a huge parking lot with only a few cars parked: a dense array writes down every spot, empty or not, while a sparse array just writes down the locations of the cars.

Deep Dive

Dense arrays are straightforward: every position has a value, even if it is zero or unused. This makes access simple and fast, but wastes memory if most positions are empty. Sparse arrays solve this by storing only the useful entries.

There are many ways to represent a sparse array:

- Dictionary/Map: store index → value pairs, ignoring empty slots.
- Coordinate list (COO): keep two lists, one for indices and one for values.

- Run-length encoding: store stretches of empty values as counts, followed by the next filled value.

The key idea is to save memory at the cost of more complex indexing. Access is no longer just arithmetic (`arr[i]`) but requires looking up in the chosen structure.

Comparison:

Representation	Memory Use	Access Speed	Good For
Dense array	High	O(1)	Data with many filled elements
Sparse (map)	Low	O(1) average	Few filled elements, random access
Sparse (list)	Very low	O(n)	Very small number of entries

Worked Example

```
# Dense representation: wastes memory for mostly empty data
dense = [0] * 20
dense[3] = 10
dense[15] = 25
print("Dense array:", dense)

# Sparse representation using dictionary
sparse = {3: 10, 15: 25}
print("Sparse array:", sparse)

# Access value
print("Value at index 3:", sparse.get(3, 0))
print("Value at index 7:", sparse.get(7, 0))  # default to 0 for missing
```

This shows how a sparse dictionary only records the positions that matter, while the dense version allocates space for all 20 slots.

Why it matters

Sparse arrays are crucial when working with large data where most entries are empty. They save memory and make it possible to process huge datasets that would not fit into memory as dense arrays. They also appear in real-world systems like machine learning (feature vectors), scientific computing (matrices with few non-zero entries), and search engines (posting lists).

Exercises

1. Represent a sparse array of size 1000 with only 3 non-zero values at indices 2, 500, and 999.
2. Write a procedure to count the number of non-empty values in a sparse array.
3. Access an index that does not exist in the sparse array and explain what should be returned.
4. Compare the memory used by a dense array of 1000 zeros and a sparse representation with 3 values.
5. Think of a real-world example (outside programming) where recording only the "non-empty" spots is more efficient than listing everything.

2.5 L1 — Sparse Arrays in Practice

Sparse arrays become important when dealing with very large datasets where only a few positions hold non-zero values. Instead of allocating memory for every element, practical implementations use compact structures to track only the occupied indices. This saves memory, but requires trade-offs in access speed and update complexity.

Deep Dive

There are several practical ways to represent sparse arrays:

1. Dictionary/Hash Map
    - Store index $\rightarrow$ value pairs.
    - Very fast random access and updates (average O(1)).
    - Memory overhead is higher because of hash structures.
2. Coordinate List (COO)
    - Keep two parallel arrays: one for indices, one for values.
    - Compact, easy to construct, but access is O(n).
    - Good for static data with few updates.
3. Compressed Sparse Row (CSR) / Compressed Sparse Column (CSC)
    - Widely used for sparse matrices.
    - Use three arrays: values, column indices, and row pointers (or vice versa).
    - Extremely efficient for matrix-vector operations.
    - Poor at dynamic updates, since compression must be rebuilt.
4. Run-Length Encoding (RLE)

- Store runs of zeros as counts, followed by non-zero entries.
- Best for sequences with long stretches of emptiness.

A comparison:

Format	Memory Use	Access Speed	Best For
Dictionary	Higher per-entry	O(1) avg	Dynamic updates, unpredictable indices
COO	Very low	O(n)	Static, small sparse sets
CSR/CSC	Compact	O(1) row scan, O(log n) col lookup	Linear algebra, scientific computing
RLE	Very compact	Sequential O(n), random slower	Time-series with long zero runs

Trade-offs:

- Dense arrays are fast but waste memory.
- Sparse arrays save memory but access/update complexity varies.
- Choice of structure depends on workload (frequent random access vs batch computation).

Use cases:

- Machine learning: sparse feature vectors in text classification or recommender systems.
- Graph algorithms: adjacency matrices for sparse graphs.
- Search engines: inverted index posting lists.
- Scientific computing: storing large sparse matrices for simulations.

Worked Example

```
# Sparse array using Python dictionary
sparse = {2: 10, 100: 50, 999: 7}

# Accessing
print("Value at 100:", sparse.get(100, 0))
print("Value at 3 (missing):", sparse.get(3, 0))

# Inserting new value
sparse[500] = 42
```

```
# Traversing non-empty values
for idx, val in sparse.items():
    print(f"Index {idx} → {val}")
```

For dense vs sparse comparison:

```
dense = [0] * 1000
dense[2], dense[100], dense[999] = 10, 50, 7
print("Dense uses 1000 slots, sparse uses", len(sparse), "entries")
```

Why it matters

Sparse arrays strike a balance between memory efficiency and performance. They let you work with massive datasets that would otherwise be impossible to store in memory. They also demonstrate the importance of choosing the right representation for the problem: a dictionary for dynamic updates, CSR for scientific kernels, or RLE for compressed logs.

Exercises

1. Represent a sparse array of length 1000 with values at indices 2, 100, and 999 using:
   a) a dictionary, and
   b) two parallel lists (indices, values).
2. Write a procedure that traverses only non-empty entries and prints them.
3. Explain why inserting a value in CSR format is more expensive than in a dictionary-based representation.
4. Compare memory usage of a dense array of length 1000 with only 5 non-zero entries against its sparse dictionary form.
5. Give two real-world scenarios where CSR is preferable to dictionary-based sparse arrays.

2.5 L2 — Sparse Arrays and Compressed Layouts in Systems

Sparse arrays are not only about saving memory; they embody deep design choices about compression, cache use, and hardware acceleration. At this level, the question is not "should I store zeros or not," but "which representation balances memory, access speed, and computational efficiency for the workload?"

Deep Dive

Several compressed storage formats exist, each tuned to different needs:

- COO (Coordinate List): Store parallel arrays for row indices, column indices, and values. Flexible and simple, but inefficient for repeated access because lookups require scanning.
- CSR (Compressed Sparse Row): Use three arrays: `values`, `col_indices`, and `row_ptr` to mark boundaries. Accessing all elements of a row is O(1), while finding a specific column in a row is O(log n) or linear. Excellent for sparse matrix-vector multiplication (SpMV).
- CSC (Compressed Sparse Column): Similar to CSR, but optimized for column operations.
- DIA (Diagonal): Only store diagonals in banded matrices. Extremely memory-efficient for PDE solvers.
- ELL (Ellpack/Itpack): Store each row padded to the same length, enabling SIMD and GPU vectorization. Works well when rows have similar numbers of nonzeros.
- HYB (Hybrid, CUDA): Combines ELL for regular rows and COO for irregular cases. Used in GPU-accelerated sparse libraries.

Performance and Complexity.

- Dictionaries/maps: O(1) average access, but higher overhead per entry.
- COO: O(n) lookups, better for incremental construction.
- CSR/CSC: excellent for batch operations, poor for insertions.
- ELL/DIA: high throughput on SIMD/GPU hardware but inflexible.

Sparse matrix-vector multiplication (SpMV) illustrates trade-offs. With CSR:

```
y[row] = Σ values[k] * x[col_indices[k]]
```

where `row_ptr` guides which elements belong to each row. The cost is proportional to the number of nonzeros, but performance is limited by memory bandwidth and irregular access to `x`.

Cache and alignment.

Compressed formats improve locality for sequential access but introduce irregular memory access patterns when multiplying or searching. Strided iteration can align with cache lines, but pointer-heavy layouts fragment memory. Padding (in ELL) improves SIMD alignment but wastes space.

Language and library implementations.

- Python SciPy: `csr_matrix`, `csc_matrix`, `coo_matrix`, `dia_matrix`.
- C++: Eigen and Armadillo expose CSR and CSC; Intel MKL provides highly optimized kernels.
- CUDA/cuSPARSE: Hybrid ELL + COO kernels tuned for GPUs.

System-level use cases.

- Large-scale PDE solvers and finite element methods.
- Graph algorithms (PageRank, shortest paths) using sparse adjacency matrices.
- Inverted indices in search engines (postings lists).
- Feature vectors in machine learning (bag-of-words, recommender systems).

Pitfalls.

- Insertion is expensive in compressed formats (requires shifting or rebuilding).
- Converting between formats (e.g., COO CSR) can dominate runtime if done repeatedly.
- A poor choice of format (e.g., using ELL for irregular sparsity) can waste memory or block vectorization.

Optimization and profiling.

- Benchmark SpMV across formats and measure achieved bandwidth.
- Profile cache misses and TLB behavior in irregular workloads.
- On GPUs, measure coalesced vs scattered memory access to judge format suitability.

Worked Example (Python with SciPy)

```
import numpy as np
from scipy.sparse import csr_matrix

# Dense 5x5 with many zeros
dense = np.array([
    [1, 0, 0, 0, 2],
    [0, 0, 3, 0, 0],
    [4, 0, 0, 5, 0],
    [0, 6, 0, 0, 0],
    [0, 0, 0, 7, 8]
])
```

```
# Convert to CSR
sparse = csr_matrix(dense)

print("CSR data array:", sparse.data)
print("CSR indices:", sparse.indices)
print("CSR indptr:", sparse.indptr)

# Sparse matrix-vector multiplication
x = np.array([1, 2, 3, 4, 5])
y = sparse @ x
print("Result of SpMV:", y)
```

This example shows how a dense matrix with many zeros can be stored efficiently in CSR. Only nonzeros are stored, and SpMV avoids unnecessary multiplications.

Why it matters

Sparse array formats are the backbone of scientific computing, machine learning, and search engines. Choosing the right format determines whether a computation runs in seconds or hours. At scale, cache efficiency, memory bandwidth, and vectorization potential matter as much as algorithmic complexity. Sparse arrays teach the critical lesson that representation is performance.

Exercises

1. Implement COO and CSR representations of the same sparse matrix and compare memory usage.
2. Write a small CSR-based SpMV routine and measure its speed against a dense implementation.
3. Explain why ELL format is efficient on GPUs but wasteful on highly irregular graphs.
4. In SciPy, convert a `csr_matrix` to `csc_matrix` and back. Measure the cost for large matrices.
5. Given a graph with 1M nodes and 10M edges, explain why adjacency lists and CSR are more practical than dense matrices.

2.6 Prefix Sums & Scans

2.6 L0 — Running Totals

A prefix sum, also called a scan, is a way of turning a sequence into running totals. Instead of just producing one final sum, we produce an array where each position shows the sum of all earlier elements. It is like keeping a receipt tape at the checkout: each item is added in order, and you see the growing total after each step.

Deep Dive

Prefix sums are simple but powerful. Given an array `[a0, a1, a2, …, an-1]`, the prefix sum array `[p0, p1, p2, …, pn-1]` is defined as:

- Inclusive scan:

  `pi = a0 + a1 + … + ai`

- Exclusive scan:

  `pi = a0 + a1 + … + ai-1`

  (with p0 = 0 by convention).

Example with array `[1, 2, 3, 4]`:

Index	Original	Inclusive	Exclusive
0	1	1	0
1	2	3	1
2	3	6	3
3	4	10	6

Prefix sums are built in a single pass, left to right. This is O(n) in time, requiring an extra array of length n to store results.

Once constructed, prefix sums allow fast range queries. For any subarray between indices `i` and `j`, the sum is:

```
sum(i..j) = prefix[j] - prefix[i-1]
```

This reduces what would be O(n) work into O(1) time per query.

Prefix sums also generalize beyond addition: they can be built with multiplication, min, max, or any associative operation.

Worked Example

```
arr = [1, 2, 3, 4, 5]

# Inclusive prefix sum
inclusive = []
running = 0
for x in arr:
    running += x
    inclusive.append(running)
print("Inclusive scan:", inclusive)

# Exclusive prefix sum
exclusive = [0]
running = 0
for x in arr[:-1]:
    running += x
    exclusive.append(running)
print("Exclusive scan:", exclusive)

# Range query using prefix sums
i, j = 1, 3  # sum from index 1 to 3 (2+3+4)
range_sum = inclusive[j] - (inclusive[i-1] if i > 0 else 0)
print("Range sum (1..3):", range_sum)
```

This program shows inclusive and exclusive scans, and how to use them to answer range queries quickly.

Why it matters

Prefix sums transform repeated work into reusable results. They make range queries efficient, reduce algorithmic complexity, and appear in countless applications: histograms, text processing, probability distributions, and parallel computing. They also introduce the idea of trading extra storage for faster queries, a common algorithmic technique.

Exercises

1. Compute the prefix sum of `[1, 2, 3, 4, 5]` by hand.
2. Show the difference between inclusive and exclusive prefix sums for `[5, 10, 15]`.
3. Use a prefix sum to find the sum of elements from index 2 to 4 in `[3, 6, 9, 12, 15]`.

4. Given a prefix sum array `[2, 5, 9, 14]`, reconstruct the original array.
5. Explain why prefix sums are more efficient than computing each subarray sum from scratch when handling many queries.

2.6 L1 — Prefix Sums in Practice

Prefix sums are a versatile tool for speeding up algorithms that involve repeated range queries. Instead of recalculating sums over and over, we preprocess the array once to create cumulative totals. This preprocessing costs O(n), but it allows each query to be answered in O(1).

Deep Dive

A prefix sum array is built by scanning the original array from left to right:

```
prefix[i] = prefix[i-1] + arr[i]
```

This produces the inclusive scan. The exclusive scan shifts everything rightward, leaving prefix[0] = 0 and excluding the current element.

The choice between inclusive and exclusive depends on application:

- Inclusive is easier for direct cumulative totals.
- Exclusive is more natural when answering range queries.

Once built, prefix sums enable efficient operations:

- Range queries: `sum(i..j) = prefix[j] - prefix[i-1]`.
- Reconstruction: the original array can be recovered with `arr[i] = prefix[i] - prefix[i-1]`.
- Generalization: the same idea works for multiplication (cumulative product), logical OR/AND, or even min/max. The key requirement is that the operation is associative.

Trade-offs:

- Building prefix sums requires O(n) extra memory.
- If only a few queries are needed, recomputing directly may be simpler.
- For many queries, the preprocessing overhead is worthwhile.

Use cases:

- Fast range-sum queries in databases or competitive programming.
- Cumulative frequencies in histograms.
- Substring analysis in text algorithms (e.g., number of vowels in a range).
- Probability and statistics: cumulative distribution functions.

Language implementations:

- Python: `itertools.accumulate`, `numpy.cumsum`.
- C++: `std::partial_sum` from `<numeric>`.
- Java: custom loop, or stream reductions.

Pitfalls:

- Confusing inclusive vs exclusive scans often leads to off-by-one errors.
- For large datasets, cumulative sums may overflow fixed-width integers.

Worked Example

```
import itertools
import numpy as np

arr = [2, 4, 6, 8, 10]

# Inclusive prefix sum using Python loop
inclusive = []
running = 0
for x in arr:
    running += x
    inclusive.append(running)
print("Inclusive prefix sum:", inclusive)

# Exclusive prefix sum
exclusive = [0]
running = 0
for x in arr[:-1]:
    running += x
    exclusive.append(running)
print("Exclusive prefix sum:", exclusive)
```

```
# NumPy cumsum (inclusive)
np_inclusive = np.cumsum(arr)
print("NumPy inclusive scan:", np_inclusive)

# Range query using prefix sums
i, j = 1, 3  # indices 1..3 → 4+6+8
range_sum = inclusive[j] - (inclusive[i-1] if i > 0 else 0)
print("Range sum (1..3):", range_sum)

# Recover original array from prefix sums
reconstructed = [inclusive[0]] + [inclusive[i] - inclusive[i-1] for i in range(1, len(inclusi
print("Reconstructed array:", reconstructed)
```

This example demonstrates building prefix sums by hand, using built-in libraries, answering queries, and reconstructing the original array.

Why it matters

Prefix sums reduce repeated work into reusable results. They transform O(n) queries into O(1), making algorithms faster and more scalable. They are a foundational idea in algorithm design, connecting to histograms, distributions, and dynamic programming.

Exercises

1. Build both inclusive and exclusive prefix sums for `[5, 10, 15, 20]`.
2. Use prefix sums to compute the sum of elements from index 2 to 4 in `[1, 3, 5, 7, 9]`.
3. Given a prefix sum array `[3, 8, 15, 24]`, reconstruct the original array.
4. Write a procedure that computes cumulative products (scan with multiplication).
5. Explain why prefix sums are more useful when answering hundreds of queries instead of just one.

2.6 L2 — Prefix Sums and Parallel Scans

Prefix sums seem simple, but at scale they become a central systems primitive. They serve as the backbone of parallel algorithms, GPU kernels, and high-performance libraries. At this level, the focus shifts from "what is a prefix sum" to "how can we compute it efficiently across thousands of cores, with minimal synchronization and maximal throughput?"

Deep Dive

Sequential algorithm. The simple prefix sum is O(n):

```
prefix[0] = arr[0]
for i in 1..n-1:
    prefix[i] = prefix[i-1] + arr[i]
```

Efficient for single-threaded contexts, but inherently sequential because each value depends on the one before it.

Parallel algorithms. Two key approaches dominate:

- Hillis–Steele scan (1986):
  - Iterative doubling method.
  - At step k, each thread adds the value from 2^k positions behind.
  - O(n log n) work, O(log n) depth. Simple but not work-efficient.
- Blelloch scan (1990):
  - Work-efficient, O(n) total operations, O(log n) depth.
  - Two phases:
    - Up-sweep (reduce): build a tree of partial sums.
    - Down-sweep: propagate sums back down to compute prefix results.
  - Widely used in GPU libraries.

Hardware performance.

- Cache-aware scans: memory locality matters for large arrays. Blocking and tiling reduce cache misses.
- SIMD vectorization: multiple prefix elements are computed in parallel inside CPU vector registers.
- GPUs: scans are implemented at warp and block levels, with CUDA providing primitives like `thrust::inclusive_scan`. Warp shuffles (`__shfl_up_sync`) allow efficient intra-warp scans without shared memory.

Memory and synchronization.

- In-place scans reduce memory use but complicate parallelization.
- Exclusive vs inclusive variants require careful handling of initial values.
- Synchronization overhead and false sharing are common risks in multithreaded CPU scans.
- Distributed scans (MPI) require combining partial results from each node, then adjusting local scans with offsets.

Libraries and implementations.

- C++ TBB: `parallel_scan` supports both exclusive and inclusive.
- CUDA Thrust: `inclusive_scan`, `exclusive_scan` for GPU workloads.
- OpenMP: provides `#pragma omp parallel for reduction` but true scans require more explicit handling.
- MPI: `MPI_Scan` and `MPI_Exscan` provide distributed prefix sums.

System-level use cases.

- Parallel histogramming: count frequencies in parallel, prefix sums to compute cumulative counts.
- Radix sort: scans partition data into buckets efficiently.
- Stream compaction: filter elements while maintaining order.
- GPU memory allocation: prefix sums assign disjoint output positions to threads.
- Database indexing: scans help build offsets for columnar data storage.

Pitfalls.

- Race conditions when threads update overlapping memory.
- Load imbalance in irregular workloads (e.g., skewed distributions).
- Wrong handling of inclusive vs exclusive leads to subtle bugs in partitioning algorithms.

Profiling and optimization.

- Benchmark sequential vs parallel scan on arrays of size 10^6 or 10^9.
- Compare scalability with 2, 4, 8, … cores.
- Measure GPU kernel efficiency at warp, block, and grid levels.

Worked Example (CUDA Thrust)

```
#include <thrust/device_vector.h>
#include <thrust/scan.h>
#include <iostream>

int main() {
    thrust::device_vector<int> data{1, 2, 3, 4, 5};

    // Inclusive scan
    thrust::inclusive_scan(data.begin(), data.end(), data.begin());
    std::cout << "Inclusive scan: ";
    for (int x : data) std::cout << x << " ";
    std::cout << std::endl;

    // Exclusive scan
    thrust::device_vector<int> data2{1, 2, 3, 4, 5};
    thrust::exclusive_scan(data2.begin(), data2.end(), data2.begin(), 0);
    std::cout << "Exclusive scan: ";
    for (int x : data2) std::cout << x << " ";
    std::cout << std::endl;
}
```

This program offloads prefix sum computation to the GPU. With thousands of threads, even huge arrays can be scanned in milliseconds.

Why it matters

Prefix sums are a textbook example of how a simple algorithm scales into a building block of parallel computing. They are used in compilers, graphics, search engines, and machine learning systems. They show how rethinking algorithms for hardware (CPU caches, SIMD, GPUs, distributed clusters) leads to new designs.

Exercises

1. Implement the Hillis–Steele scan for an array of length 16 and show each step.
2. Implement the Blelloch scan in pseudocode and explain how the up-sweep and down-sweep phases work.
3. Benchmark a sequential prefix sum vs an OpenMP parallel scan on 10^7 elements.
4. In CUDA, implement an exclusive scan at the warp level using shuffle instructions.

5. Explain how prefix sums are used in stream compaction (removing zeros from an array while preserving order).

Deep Dive

2.1 Static Arrays

- Memory alignment and padding in C and assembly.
- Array indexing formulas compiled into machine code.
- Page tables and kernel use of fixed-size arrays (`task_struct`, `inode`).
- Vectorization of loops over static arrays (SSE/AVX).
- Bounds checking elimination in high-level languages.

2.2 Dynamic Arrays

- Growth factor experiments: doubling vs 1.5× vs incremental.
- Profiling Python's list growth strategy (measure capacity jumps).
- Amortized vs worst-case complexity: proofs with actual benchmarks.
- Reallocation latency spikes in low-latency systems.
- Comparing `std::vector::reserve` vs default growth.
- Memory fragmentation in long-running programs.

2.3 Slices & Views

- Slice metadata structure in Go (`ptr, len, cap`).
- Rust borrow checker rules for `&[T]` vs `&mut [T]`.
- NumPy stride tricks: transpose as a view, not a copy.
- Performance gap: traversing contiguous vs strided slices.
- Cache/TLB impact of strided access (e.g., step=16).
- False sharing when two threads use overlapping slices.

2.4 Multidimensional Arrays

- Row-major vs column-major benchmarks: traverse order timing.
- Linear index formulas for N-dimensional arrays.
- Loop tiling/blocking for matrix multiplication.
- Structure of Arrays (SoA) vs Array of Structures (AoS).
- False sharing and padding in multi-threaded traversal.
- BLAS/LAPACK optimizations and cache-aware kernels.

- GPU coalesced memory access in 2D/3D arrays.

2.5 Sparse Arrays & Compressed Layouts

- COO, CSR, CSC: hands-on with memory footprint and iteration cost.
- Comparing dictionary-based vs CSR-based sparse vectors.
- Parallel SpMV benchmarks on CPU vs GPU.
- DIA and ELL formats: why they shine in structured sparsity.
- Hybrid GPU formats (HYB: ELL + COO).
- Search engine inverted indices as sparse structures.
- Sparse arrays in ML: bag-of-words and embeddings.

2.6 Prefix Sums & Scans

- Inclusive vs exclusive scans: correctness pitfalls.
- Hillis–Steele vs Blelloch scans: step count vs work efficiency.
- Cache-friendly prefix sums on CPUs (blocked scans).
- SIMD prefix sum using AVX intrinsics.
- CUDA warp shuffle scans (`__shfl_up_sync`).
- MPI distributed scans across clusters.
- Stream compaction via prefix sums (remove zeros in O(n)).
- Radix sort built from parallel scans.

LAB

2.1 Static Arrays

- LAB 1: Implement fixed-size arrays in C and Python, compare access/update speeds.
- LAB 2: Explore how static arrays are used in Linux kernel (`task_struct`, page tables).
- LAB 3: Disassemble a simple loop over a static array and inspect the generated assembly.
- LAB 4: Benchmark cache effects: sequential vs random access in a large static array.

2.2 Dynamic Arrays

- LAB 1: Implement your own dynamic array in C (with doubling strategy).
- LAB 2: Benchmark Python's `list` growth by tracking capacity changes while appending.
- LAB 3: Compare growth factors: doubling vs 1.5× vs fixed increments.
- LAB 4: Stress test reallocation cost by appending millions of elements, measure latency spikes.

- LAB 5: Use `std::vector::reserve` in C++ and compare performance vs default growth.

2.3 Slices & Views

- LAB 1: In Go, experiment with slice creation, capacity, and append — observe when new arrays are allocated.
- LAB 2: In Rust, create overlapping slices and see how the borrow checker enforces safety.
- LAB 3: In Python, compare slicing a list vs slicing a NumPy array — demonstrate copy vs view behavior.
- LAB 4: Benchmark stride slicing in NumPy (`arr[::16]`) and explain performance drop.
- LAB 5: Demonstrate aliasing bugs when two slices share the same underlying array.

2.4 Multidimensional Arrays

- LAB 1: Write code to traverse a 1000×1000 array row by row vs column by column, measure performance.
- LAB 2: Implement your own 2D array in C using both contiguous memory and array-of-pointers, compare speed.
- LAB 3: Use NumPy to confirm row-major order with `.strides`, then create a column-major array and compare.
- LAB 4: Implement a tiled matrix multiplication in C/NumPy and measure cache improvement.
- LAB 5: Experiment with SoA vs AoS layouts for a struct of 3 floats (x,y,z). Measure iteration performance.

2.5 Sparse Arrays & Compressed Layouts

- LAB 1: Implement sparse arrays with Python dict vs dense lists, compare memory usage.
- LAB 2: Build COO and CSR representations for the same matrix, print memory layout.
- LAB 3: Benchmark dense vs CSR matrix-vector multiplication.
- LAB 4: Use SciPy's `csr_matrix` and `csc_matrix`, run queries, compare performance.
- LAB 5: Implement a simple search engine inverted index as a sparse array of word→docID list.

2.6 Prefix Sums & Scans

- LAB 1: Write inclusive and exclusive prefix sums in Python.
- LAB 2: Benchmark prefix sums for answering 1000 range queries vs naive summation.
- LAB 3: Implement Blelloch scan in C/NumPy and visualize the up-sweep/down-sweep steps.

- LAB 4: Implement prefix sums on GPU (CUDA/Thrust), compare speed to CPU.
- LAB 5: Use prefix sums for stream compaction: remove zeros from an array while preserving order.