The Little Book of Algorithms

Version 0.1.2

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)

```
# 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)

```
# 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.

Chapter 3. Strings

3.1 Representation

3.1 L0 — Necklace of Characters

A string is a sequence of characters. Each character can be a letter, number, symbol, or even whitespace. Strings are used to represent text, from simple names and messages to entire documents. In programming, a string is usually enclosed in quotes to tell the computer where it begins and ends. Strings are fundamental because almost every program interacts with text: input from users, output on screens, or data stored in files.

Deep Dive

Strings look simple but have a few important properties that beginners must understand.

In many languages such as Python, strings cannot be changed after creation. If a program "changes" a string, what really happens is the creation of a new one. This is why appending repeatedly can be costly.

Each character in a string has a position, starting at 0. Accessing is direct: `word[0]` gives the first character. Slicing extracts portions: `word[1:4]` creates a substring.

String	Index Positions
`"HELLO"`	H=0, E=1, L=2, L=3, O=4

Negative indices count backward: `word[-1]` gives the last character.

Strings can be combined or repeated. Concatenation uses the `+` operator; repetition uses `*`. Example: `"Hi" + "!"` → `"Hi!"`, `"Na" * 4` → `"NaNaNaNa"`.

Quotes (`'` or `"`) mark strings. Escape characters handle special cases: `"\n"` is newline, `"\t"` is tab. Triple quotes allow multi-line text.

Computers store strings as numbers. ASCII is a small table of 128 characters (A–Z, digits, symbols). Unicode is a universal table covering all writing systems. Modern programs use Unicode (usually UTF-8 encoding), allowing text like "　" or "🙂".

Worked Example (Python)

```
# Defining different kinds of strings
s1 = "Hello"
s2 = 'World'
s3 = """This is
a multi-line string."""

# Indexing
print(s1[0])      # H
print(s1[-1])     # o

# Slicing
print(s1[1:4])    # ell

# Concatenation and repetition
greeting = s1 + " " + s2
print(greeting)   # Hello World

laugh = "Ha" * 3
print(laugh)      # HaHaHa

# Escapes
quote = "She said: \"Yes!\""
print(quote)      # She said: "Yes!"

# Unicode characters
emoji = "Smile  "
print(emoji)      # Smile
```

Why it matters

Strings are everywhere. They store names, labels, and text messages. They carry commands and configurations in programs. They are the backbone of web pages, logs, and data files. Understanding immutability avoids performance mistakes. Knowing slicing and concatenation makes manipulation easier. Being aware of Unicode prevents bugs in multilingual applications. Strings look simple but are a core abstraction every developer must master.

Exercises

1. Take the word `"COMPUTER"` and print the first, middle, and last character.
2. Slice the string `"PROGRAMMING"` to extract `"GRAM"`.
3. Concatenate `"DATA"` and `"BASE"` into one string without using `+`.
4. Repeat the string `"Na"` four times, then add `" Batman!"` at the end.
5. Create a multi-line string containing your name, age, and favorite color.
6. Find the difference between `word[0]` and `word[-1]` in `"PYTHON"`.
7. Write a string that contains both single and double quotes.
8. Show that `"abc" * 0` results in an empty string.
9. Print the Unicode string for `" "` alongside a normal word.
10. Explain in one sentence why immutability matters when working with strings.

3.1 L1 — Encodings & Efficiency

Strings in modern programming languages are more than just "text." They are data structures designed with specific trade-offs in memory layout, immutability, and performance. At this level, understanding how strings behave internally allows programmers to write faster, safer, and more maintainable code.

Overview/Definition

A string is immutable in many languages, meaning once created, it cannot be changed. This design improves safety and makes strings easier to reason about, but it also affects performance. Operations like concatenation, slicing, and encoding conversions create new strings under the hood. Intermediate programmers must understand how these costs accumulate, when interning optimizes memory usage, and how encodings like UTF-8 and UTF-16 impact representation.

Deep Dive

Strings cannot be modified in place in Python, Java, or Go. If `"hello"` is changed to `"hallo"`, the original object remains untouched, and a new one is created. This has two main consequences:

- Safe sharing across code without fear of accidental modification.
- Extra memory and CPU costs when building new strings repeatedly.

Languages often reuse identical string values. For example, Python keeps a single `"foo"` literal in memory if used multiple times. This is called *interning*.

- Automatic interning happens for identifiers and some literals.

- Manual interning (`sys.intern()`) can reduce memory in applications with repeated keys (like symbol tables or parsers).

Strings store characters differently across languages:

- UTF-8: Variable-width, 1–4 bytes per character. Compact for ASCII-heavy text.
- UTF-16: Mostly 2 bytes per character, but uses surrogate pairs for others.
- UTF-32: Fixed 4 bytes per character, simple but wasteful.

Encoding	Example Text `"A "`	Storage Size
ASCII	not representable	—
UTF-8	`0x41 0xF0 0x9F 0x98 0x8A`	5 bytes
UTF-16	`0x0041 0xD83D 0xDE0A`	6 bytes
UTF-32	`0x00000041 0x0001F60A`	8 bytes

This matters because operations like slicing or indexing depend on how characters are encoded.

Concatenation Costs

- In Python, `a + b` builds a new string by copying both inputs.
- Repeated concatenation inside loops leads to quadratic performance.
- Solution: accumulate pieces in a list and join once.

Example of inefficient vs efficient approach:

```
# Inefficient
result = ""
for i in range(1000):
    result += str(i)

# Efficient
parts = []
for i in range(1000):
    parts.append(str(i))
result = "".join(parts)
```

Slicing and Substrings

- In Python, `s[2:5]` creates a new string, copying data.

- In Java before version 7u6, substrings shared the same underlying array (risking memory leaks if a small substring referenced a huge array). After 7u6, substring copies data to avoid this issue.
- In C++, `std::string_view` allows non-owning references to substrings without copying.

Encoding and Decoding

- Strings are abstract characters; at some point, they must become bytes.
- `.encode()` turns a Python `str` into bytes using a given encoding.
- `.decode()` reverses the process.
- Pitfalls: decoding with the wrong encoding leads to errors or corrupted text.

Performance Pitfalls in Practice

- Parsing logs or JSON with millions of lines can cause memory spikes if string copies accumulate.
- Unicode introduces complexity: slicing `" "` may split it in half if treated as bytes instead of characters.
- Benchmarking is essential when dealing with high-throughput text pipelines.

Cross-Language Comparisons

Language	Representation	Notes
Python	Unicode (PEP 393 flexible storage)	Optimized for compactness.
Java	UTF-16, immutable `String`	Uses `StringBuilder` for mutable ops.
C++	`std::string`, `string_view`	SSO (small string optimization) avoids heap allocation for short strings.
Go	UTF-8 encoded immutable slices	`string` is a read-only `[]byte`.

Worked Example (Python)

```
# Interning demonstration
import sys

a = "hello"
b = "hello"
print(a is b)  # True, both refer to the same interned literal
```

```
# Manual interning
x = sys.intern("repeated_key")
y = sys.intern("repeated_key")
print(x is y)  # True

# Encoding and decoding
text = "Café"
encoded = text.encode("utf-8")
print(encoded)  # b'Caf\xc3\xa9'
decoded = encoded.decode("utf-8")
print(decoded)  # Café

# Concatenation efficiency
words = ["alpha", "beta", "gamma"]
joined = "-".join(words)
print(joined)  # alpha-beta-gamma
```

Why it matters

Intermediate-level understanding of string representation prevents subtle but costly mistakes. Knowing that strings are immutable avoids slow concatenation loops. Interning saves memory in large-scale text-heavy systems. Choosing the right encoding prevents bugs when handling international text. Understanding substring behavior helps avoid hidden memory leaks. These details directly impact performance, correctness, and reliability in production systems.

Exercises

1. Write code that shows the difference in performance between concatenating strings in a loop and using `join`.
2. Demonstrate string interning: show two identical literals referencing the same object.
3. Encode and decode a string containing accented characters using UTF-8.
4. Take a long string and slice it into two halves; explain why slicing creates a new object in Python.
5. Compare memory usage between UTF-8 and UTF-16 for a text that contains only ASCII characters.
6. Investigate what happens when decoding a UTF-8 byte sequence with ASCII encoding.
7. Show why `" "[0]` in Python gives a character but in UTF-8 byte arrays it spans multiple bytes.
8. Explain the difference between Java `String` and `StringBuilder` in terms of mutability.
9. Write code that demonstrates the effect of `sys.intern()` on repeated keys.

10. Research and explain how `std::string_view` in C++ avoids copies.

3.1 L2 — Internals & Systems

Strings are central to modern software, but their internal design depends heavily on language, runtime, and operating system. At this level, understanding low-level representation, OS interaction, and hardware acceleration becomes critical.

Strings are immutable in most high-level languages, but under the hood they are arrays of bytes or characters managed by the runtime. Different languages make different trade-offs: memory layout, encoding choice, and substring handling. The operating system provides system calls for moving string data between memory and devices. Hardware accelerates common operations like copying and comparison. Production libraries extend functionality for correctness and speed.

Deep Dive

Low-Level Representations

C strings are arrays of `char` terminated by `'\0'`. A missing terminator causes buffer overflows. C++ `std::string` stores length and data, with small string optimization (SSO) to keep short strings inline. Python uses PEP 393: compact Unicode with 1, 2, or 4 bytes per character depending on the highest code point. Java `String` is UTF-16, compressed if all characters fit in Latin-1. Go strings are immutable slices of bytes; Rust distinguishes `String` (owned) and `&str` (borrowed).

Language	Representation	Notes
C	`char *` + `'\0'`	Simple but unsafe.
C++	`std::string`, SSO	Short strings inline.
Python	Flexible Unicode	Adapts width per string.
Java	UTF-16, compressed	Immutable.
Go	UTF-8, slice view	Immutable.
Rust	`String` / `&str`	UTF-8 guaranteed.

OS-Level Considerations

Strings occupy stack or heap memory. Large strings cause fragmentation. Garbage-collected languages like Java and Python reclaim unused strings automatically. System calls (`read`, `write`) exchange raw bytes with the OS. In C, forgetting `'\0'` causes overruns. At boundaries (sockets, files), encoding mismatches are a common source of bugs.

Hardware-Level Considerations

Performance depends on cache alignment. A string aligned to cache lines is scanned faster. Libraries like `glibc` use SIMD instructions for `memcpy`, `memcmp`, and `strlen`, processing multiple bytes per CPU instruction. Large-scale search and comparison rely on vectorized instructions. Misaligned memory leads to extra CPU cycles.

Advanced Encoding Issues

Unicode normalization ensures two strings that look identical are stored in the same form. NFC and NFD differ in how accents are represented. UTF-16 surrogate pairs represent code points beyond `0xFFFF`. Grapheme clusters (like “ ”) span multiple code points but behave as one character for users. Security issues arise: Unicode confusables trick users (in Cyrillic vs `a` in Latin), and null characters may bypass string checks in C.

Production Libraries & Techniques

ICU provides collation, normalization, and locale-sensitive operations. RE2 avoids regex backtracking vulnerabilities. Hyperscan accelerates string matching using SIMD. String views or slices in C++ and Rust enable zero-copy substrings.

Performance Engineering

Concatenation cost is linear in total length. Substrings may leak memory if they reference a large parent buffer (Java pre-7u6). Copy semantics avoid leaks but cost memory. Benchmarking is necessary to choose the right trade-off. Immutability simplifies reasoning but requires builders (`StringBuilder`, `bytes.Buffer`) for efficiency.

Worked Example (Python)

```
import sys
import unicodedata
import time

# --- Representation ---
ascii_text = "Hello"
unicode_text = "    "
print("ASCII:", len(ascii_text), "chars,", len(ascii_text.encode("utf-8")), "bytes")
print("Unicode:", len(unicode_text), "chars,", len(unicode_text.encode("utf-8")), "bytes")

# --- Encoding/Decoding ---
```

```
data = "Café"
utf8_bytes = data.encode("utf-8")
print("UTF-8 bytes:", utf8_bytes)
print("Decoded back:", utf8_bytes.decode("utf-8"))

# --- Interning ---
a = "hello"
b = "hello"
print("a is b:", a is b)
x = sys.intern("repeated_key")
y = sys.intern("repeated_key")
print("x is y:", x is y)

# --- Concatenation benchmark ---
N = 20000
start = time.time()
s = ""
for i in range(N):
    s += "x"
print("Naive concat time:", round(time.time() - start, 4), "s")

start = time.time()
parts = ["x" for _ in range(N)]
s = "".join(parts)
print("Join concat time:", round(time.time() - start, 4), "s")

# --- Unicode normalization ---
s1 = "café"          # composed
s2 = "cafe\u0301"    # decomposed
print("Raw equal:", s1 == s2)
print("NFC equal:", unicodedata.normalize("NFC", s1) == unicodedata.normalize("NFC", s2))

# --- Surrogate pairs & grapheme clusters ---
smile = " "
print("UTF-16 units:", len(smile.encode("utf-16")) // 2)
family = "  "
print("Family code points:", len(family), "rendered:", family)

# --- Unicode confusables ---
latin_a = "a"
cyrillic_a = " "  # visually similar
print("Latin a == Cyrillic a:", latin_a == cyrillic_a)
```

```
print("Latin ord:", ord(latin_a), "Cyrillic ord:", ord(cyrillic_a))
```

Why it matters

Knowing internals prevents bugs and security issues. Buffer overflows from missing terminators break systems. Misunderstood Unicode can cause errors in databases or user interfaces. Cache alignment and SIMD accelerate processing of huge text datasets. Normalization avoids mismatched text values. Confusables create attack vectors. Production systems depend on engineers understanding these low-level details.

Exercises

1. Explain why a string with a missing terminator can cause reading beyond its memory.
2. Compare how many bytes the string `"Hello"` uses in UTF-8, UTF-16, and UTF-32.
3. Show two visually identical strings that are not equal because of different Unicode code points.
4. Demonstrate normalization making two unequal strings equal.
5. Measure and compare performance between naive concatenation and buffered concatenation.
6. Explain how surrogate pairs represent characters beyond `0xFFFF`.
7. Construct a grapheme cluster (like a family emoji) and count its code points versus user-perceived characters.
8. Show how substring references can create memory leaks if the original large string is kept alive.
9. Investigate how cache alignment could affect the speed of scanning a very large string.
10. Design a test case that reveals a security risk using Unicode confusables.

3.2 Operations

3.2 L0 — Everyday String Manipulations

Strings are not only stored but also actively used and modified. Beginners often start with everyday tasks: changing text to uppercase, trimming spaces, searching for words, and building sentences. These operations are simple yet powerful, and they form the foundation of text processing in any program.

String operations are ways to manipulate or examine text. They include changing case, removing whitespace, searching for substrings, splitting sentences into words, joining words into sentences, and formatting messages. These actions are essential for handling user input, displaying results, and working with text files.

Deep Dive

Case Conversion

Text often needs to be converted. Uppercase makes words stand out (`"hello" → "HELLO"`). Lowercase is useful for comparisons (`"Yes"` vs `"yes"`). Case conversion helps normalize text before processing.

Trimming Whitespace

Whitespace before or after text causes bugs when comparing values. `" hello "` is not the same as `"hello"`. Trimming removes unwanted spaces, tabs, or newlines.

Replacing Substrings

Sometimes one part of text must change. `"cat".replace("c", "h")` becomes `"hat"`. Replacing works for single characters or words.

Searching in Strings

Finding a substring answers whether `"dog"` appears inside `"hotdog"`. Many languages return the index where the substring begins. If not found, they return a special value (like `-1` or `None`).

Operation	Example	Result
Contains	`"hello" in "say hello"`	True
Index	`"abc".find("b")`	1
Not found	`"abc".find("z")`	-1 (or equivalent)

Splitting and Joining

Splitting breaks text into parts: `"one two three"` → `["one", "two", "three"]`. Joining does the reverse: `["a","b","c"]` → `"a-b-c"`. They are inverse operations.

Formatting

Combining variables with text creates messages. `"Hello " + name` works, but formatting systems are clearer: `"Hello, {name}!"`. Formatting ensures readable and consistent text output.

Worked Example (Python)

```python
# Case conversion
word = "Hello"
print(word.upper())   # HELLO
print(word.lower())   # hello

# Trimming whitespace
text = "   data  "
print("Before:", repr(text))
print("After:", repr(text.strip()))

# Replacing substrings
animal = "cat"
print(animal.replace("c", "h"))  # hat

# Searching
sentence = "the quick brown fox"
print("fox" in sentence)        # True
print(sentence.find("quick"))   # 4
print(sentence.find("dog"))     # -1

# Splitting and joining
line = "one two three"
parts = line.split(" ")
print(parts)  # ['one', 'two', 'three']
joined = "-".join(parts)
print(joined)  # one-two-three

# Formatting
name = "Alice"
age = 30
print(f"My name is {name}, I am {age} years old.")
```

Why it matters

Text is everywhere: names, messages, configuration files, logs. Programs must read, search, and reshape strings constantly. Converting case enables reliable comparisons. Trimming prevents subtle errors with extra spaces. Splitting and joining support structured data handling. Formatting improves readability for users. These basic operations turn raw text into structured, usable information.

Exercises

1. Convert the string `"python"` to uppercase and lowercase.
2. Remove extra spaces from `"   hello world   "`.
3. Replace `"dog"` with `"cat"` in the sentence `"the dog barked"`.
4. Check if `"apple"` is inside `"pineapple"`.
5. Find the position of `"moon"` in `"the moon rises"`.
6. Split `"red,green,blue"` into parts and then join them back with `"|"`.
7. Count how many times the word `"the"` appears in `"the cat and the dog"`.
8. Format a string that says `"My favorite number is X"`, where `X` is any integer.
9. Combine a list of words into a sentence with spaces in between.
10. Explain in one sentence why trimming whitespace is important when processing user input.

Do you want me to continue with 3.2 L1 in the same structured style, adding intermediate concepts like regex basics and efficient searching?

3.2 L1 — Patterns & Practical Tricks

Strings are not only for simple manipulations. In real programs, text must be searched, parsed, and transformed efficiently. At this level, the focus shifts from basic usage to techniques that handle larger inputs, avoid performance traps, and apply structured text processing.

Intermediate string operations expand beyond uppercase, trimming, and splitting. They involve efficient searching, counting, and pattern matching. Regular expressions become a tool for flexible matching. Developers also need to parse structured data such as CSV and JSON, where correctness and performance both matter.

Deep Dive

Efficient Searching

Most languages provide methods like `find`, `index`, `startswith`, and `endswith`. These run in linear time but are implemented with optimizations in native code. Checking prefixes and suffixes avoids scanning the entire string.

Counting and Replacing

Counting how many times a substring occurs is essential for statistics or validation. Replacing substrings can transform logs or templates. Efficiency matters: repeated replacements in large text can be costly.

Operation	Example Input	Result
Startswith	`"hello".startswith("he")`	True
Endswith	`"world".endswith("ld")`	True
Count substring	`"banana".count("na")`	2
Replace	`"2025-09".replace("-", "/")`	`"2025/09"`

Regular Expressions (Regex)

Regex allows powerful pattern matching with symbols:

- `.` matches any character.
- `*` means repeat zero or more times.
- `+` means one or more times.
- `[abc]` matches any of the listed characters.

Example:

- Pattern `[0-9]+` matches any sequence of digits.
- Pattern `\w+@\w+\.\w+` matches simple email addresses.

Regex engines vary: some use backtracking (flexible but can be slow), others use DFA/NFA (linear-time but less expressive).

Parsing Structured Text

Many formats (CSV, JSON, logs) are line-based text. Splitting on commas or spaces is not enough for robust parsing, but for simple cases splitting and trimming work. Intermediate developers should know the limitations of naive parsing and when to use libraries.

Performance Pitfalls

Naive string concatenation inside loops can be quadratic in cost. Multiple find/replace calls on large inputs can also degrade performance. The best practice is to buffer results or use streaming parsers.

Worked Example (Python)

```python
import re

text = "user: alice, email: alice@example.com; user: bob, email: bob@example.org"

# Efficient searching
print(text.startswith("user"))      # True
print(text.endswith("org"))         # True
print(text.find("bob"))             # position of 'bob'

# Counting and replacing
fruit = "banana"
print(fruit.count("na"))            # 2
print(fruit.replace("na", "NA"))    # baNANA

# Regex: find all email addresses
emails = re.findall(r"\w+@\w+\.\w+", text)
print(emails)  # ['alice@example.com', 'bob@example.org']

# Regex: extract all numbers from a string
data = "Order 123, item 456, total 789"
numbers = re.findall(r"[0-9]+", data)
print(numbers)  # ['123', '456', '789']

# Parsing structured text (simple CSV line)
line = "apple, banana, cherry"
parts = [p.strip() for p in line.split(",")]
print(parts)  # ['apple', 'banana', 'cherry']

# Performance demo: avoid naive concatenation
N = 10000
# Slow
s = ""
for i in range(N):
    s += "x"
print("Naive length:", len(s))

# Fast
s = "".join(["x" for _ in range(N)])
print("Join length:", len(s))
```

Why it matters

Intermediate operations turn raw text into structured data. Searching and counting enable validation and analysis. Regex provides flexible pattern extraction, but misuse can harm performance. Parsing prepares text for further computation. Avoiding pitfalls like naive concatenation ensures scalable code. These techniques are essential for handling logs, configurations, user inputs, and datasets in real-world applications.

Exercises (Easy → Hard)

1. Check if the string `"algorithm"` starts with `"algo"` and ends with `"rithm"`.
2. Count how many times `"the"` appears in `"the cat and the dog and the theater"`.
3. Replace every `"-"` in `"2025-09-10"` with `"/"`.
4. Extract all numbers from the text `"room 101, floor 5, building 42"`.
5. Write a regex to match any word that starts with `"a"` and ends with `"e"`.
6. Parse `"red, green, blue , yellow"` into a clean list of color names without spaces.
7. Demonstrate why repeated concatenation in a loop is slower than using a buffer or join.
8. Write a regex to match simple phone numbers like `"123-456-7890"`.
9. Explain why splitting CSV by commas fails if values contain quotes (e.g., `"apple, "banana,grape", cherry"`).
10. Design a regex that extracts domain names from email addresses.

3.2 L2 — Algorithms & Systems

Advanced string operations rely on specialized algorithms and system support. Searching and matching must be efficient on gigabytes of text. Regex engines balance flexibility with safety. Compilers tokenize source code into symbols using string scanning. And at the system boundary, encoding and locale awareness become critical for correct communication.

Deep Dive

String Search Algorithms

The naive search scans one character at a time, which is simple but inefficient for large text.

- Knuth–Morris–Pratt (KMP) preprocesses the pattern to avoid redundant comparisons.
- Boyer–Moore skips ahead using knowledge of mismatched characters, performing sublinear searches in practice.
- These algorithms form the backbone of text editors, search utilities, and compilers.

Regex Engine Internals

Regex engines are either backtracking-based (Perl, Python) or automata-based (RE2, Rust).

- Backtracking engines support rich features but risk exponential slowdowns.
- DFA/NFA-based engines guarantee linear-time execution but restrict advanced patterns.
- JIT compilation of regex (like in Java or .NET) generates machine code for speed.

Tokenization and Lexical Analysis

Compilers and interpreters must scan source code into tokens. This uses deterministic finite automata (DFAs) built from regex-like rules. For example, recognizing identifiers, keywords, and numbers all rely on string scanning.

Unicode-Aware Searching

Searching in Unicode text requires more than simple byte comparison. Case folding (making case-insensitive comparisons) differs by locale. Grapheme clusters must be considered as user-visible units, not code points.

System Boundaries

Strings cross boundaries when written to files, sockets, or system calls. Encodings must match between sender and receiver. Misalignment causes corrupted output or runtime errors. Databases and network APIs rely on normalized, correctly encoded strings.

Performance Considerations

Substring extraction may copy data (safe but costly) or reference parent buffers (fast but risky). SIMD instructions accelerate scanning large strings in modern CPUs. Benchmarks show naive operations become bottlenecks at scale, requiring optimized algorithms.

Worked Example (Python)

```
import re
import time

# --- Naive search ---
def naive_search(text, pattern):
    for i in range(len(text) - len(pattern) + 1):
        if text[i:i+len(pattern)] == pattern:
```

```
            return i
    return -1

# --- KMP search ---
def kmp_table(pattern):
    table = [0] * len(pattern)
    j = 0
    for i in range(1, len(pattern)):
        while j > 0 and pattern[i] != pattern[j]:
            j = table[j-1]
        if pattern[i] == pattern[j]:
            j += 1
            table[i] = j
    return table

def kmp_search(text, pattern):
    table = kmp_table(pattern)
    j = 0
    for i in range(len(text)):
        while j > 0 and text[i] != pattern[j]:
            j = table[j-1]
        if text[i] == pattern[j]:
            j += 1
        if j == len(pattern):
            return i - j + 1
    return -1

# --- Performance test ---
text = "a" * 100000 + "b"
pattern = "a" * 1000 + "b"

start = time.time()
print("Naive:", naive_search(text, pattern))
print("Naive time:", round(time.time() - start, 4), "s")

start = time.time()
print("KMP:", kmp_search(text, pattern))
print("KMP time:", round(time.time() - start, 4), "s")

# --- Regex engines ---
data = "user: alice@example.com id: 42"
emails = re.findall(r"\w+@\w+\.\w+", data)
```

```
print("Regex found:", emails)

# --- Unicode case folding ---
s1 = "Straße"
s2 = "STRASSE"
print("Casefold equal:", s1.casefold() == s2.casefold())

# --- Grapheme cluster awareness ---
family = "  "
print("Code points:", len(family))     # length in code points
print("Rendered:", family)             # perceived as one emoji
```

Why it matters

At scale, naive methods fail. Efficient algorithms like KMP and Boyer–Moore make search practical in editors, search engines, and compilers. Regex engines must be chosen carefully: flexible backtracking engines can crash under malicious input, while DFA-based engines provide safety. Unicode awareness ensures correctness in multilingual systems. At system boundaries, encoding mismatches corrupt data. Performance engineering determines whether systems handle megabytes or terabytes of text reliably.

Exercises

1. Describe how the naive substring search works and why it can be inefficient.
2. Compare the number of steps needed to search `"aaaaab"` in `"a"*1000 + "b"` using naive vs KMP.
3. Explain the key idea behind Boyer–Moore that makes it faster than naive search.
4. Show why some regex patterns can lead to exponential backtracking.
5. Write a regex that matches valid identifiers (letters, digits, underscores, not starting with a digit).
6. Explain how compilers use finite automata to tokenize code.
7. Compare string equality with and without Unicode case folding (`"Straße"` vs `"STRASSE"`).
8. Show how a grapheme cluster (like family emoji) differs from code point count.
9. Explain why substring references in some languages (e.g., Java before 7u6) could cause memory leaks.
10. Design a benchmark plan to measure the impact of SIMD acceleration on string scanning.

3.3 Comparison

3.3 L0 — Equality & Ordering

Comparing strings is one of the simplest but most important operations in programming. Equality checks determine if two pieces of text are the same, while ordering lets us sort words alphabetically. These operations are intuitive but have precise rules that beginners must understand.

Overview/Definition

Two strings are equal if they contain exactly the same sequence of characters in the same order. Ordering compares strings lexicographically, meaning character by character from left to right, similar to how words are sorted in a dictionary. These rules allow programs to check conditions, filter text, and sort lists.

Deep Dive

Equality

Equality is strict: `"cat"` equals `"cat"` but not `"Cat"`. Whitespace matters too, so `"hello "` is not the same as `"hello"`. Computers check equality by comparing each character in order until a difference is found or both end.

Inequality

If two strings differ in length or characters, they are not equal. The result is simply `true` or `false`.

Lexicographic Ordering Ordering compares based on the numerical values of characters. `"apple"` comes before `"banana"` because `'a'` is less than `'b'`. If the first characters are equal, the comparison moves to the next.

Comparison	Example	Result
Equal	`"cat" == "cat"`	True
Case-sensitive	`"cat" == "Cat"`	False
Shorter vs longer	`"car" < "cart"`	True
Alphabetical	`"apple" < "banana"`	True

Sorting

Sorting a list of strings uses lexicographic rules. The result is the same as dictionary order in English when restricted to simple lowercase letters.

Worked Example (Python)

```
# Equality
print("cat" == "cat")     # True
print("cat" == "Cat")     # False
print("hello " == "hello")# False

# Inequality
print("dog" != "cat")     # True

# Lexicographic ordering
print("apple" < "banana") # True
print("zebra" > "yak")    # True
print("car" < "cart")     # True

# Sorting a list
words = ["banana", "apple", "cherry"]
print(sorted(words))      # ['apple', 'banana', 'cherry']
```

Why it matters

Equality and ordering are the foundation of text handling. Programs must compare passwords, search for keywords, or sort lists of names. Without correct comparison, search engines would mis-rank results and databases would fail to find matches. Even small differences like uppercase vs lowercase or extra spaces can change outcomes. Understanding these rules prevents errors in everyday programming.

Exercises

1. Check if `"dog"` equals `"Dog"`.
2. Compare `"sun"` and `"moon"` and decide which comes first alphabetically.
3. Show why `"car"` is less than `"cart"`.
4. Sort the list `["pear", "apple", "orange"]` in dictionary order.
5. Explain why `"hello "` and `"hello"` are not equal.
6. Compare `"Zoo"` and `"apple"` and explain why uppercase letters can affect order.

7. Demonstrate that two strings must have the same length and characters to be equal.
8. Given a list of names, describe how to remove duplicates using string equality.
9. Explain what happens if two strings are equal up to the length of the shorter one (e.g., `"abc"` vs `"abcd"`).
10. Design a procedure to sort a list of mixed short and long strings alphabetically.

3.3 L1 — Collation & Locales

Comparing strings gets more complicated once you move beyond simple equality and dictionary order. Real-world applications must deal with case sensitivity, cultural rules, and mixed content like numbers and letters. At this level, programmers learn practical techniques for handling comparisons in everyday systems.

Collation is the set of rules that define how strings are compared and sorted. While lexicographic order works for plain ASCII, different languages and contexts require more sophisticated rules. For example, `"ä"` may be treated as equal to `"a"` in one locale but as a separate letter in another. Intermediate-level comparison also introduces the idea of ignoring case and whitespace when appropriate.

Deep Dive

Case Sensitivity

By default, string comparison is case-sensitive: `"cat"` ≠ `"Cat"`. Case-insensitive comparison requires converting both sides to the same case or using a locale-aware function.

Locales and Cultural Rules

Different languages define alphabetical order differently.

- In German, `"ä"` often sorts like `"ae"`.
- In Swedish, `"ä"` is a separate letter after `"z"`.
- In English, `"apple" < "Banana"` because uppercase and lowercase letters use different numeric codes, but in many systems, case-insensitive comparison is preferred.

Numbers in Strings

Lexicographic order can be confusing with numbers. `"file10"` comes before `"file2"` because `"1"` sorts before `"2"`. Natural sorting treats numbers as whole values so `"file2" < "file10"`.

Databases and Collation

Databases define collations to decide how text is compared and sorted. A collation can be case-sensitive (`CS`) or case-insensitive (`CI`). It can also be accent-sensitive (`AS`) or accent-insensitive (`AI`). This ensures consistent results across queries.

Comparison Type	Example	Result
Case-sensitive	`"cat" == "Cat"`	False
Case-insensitive	`"cat" ~ "Cat"`	True
Locale (German)	`"äpfel" vs "apfel"`	Equal or ordered together
Natural sorting	`"file2" < "file10"`	True

Worked Example (Python)

```python
import locale
import re

# Case-sensitive vs case-insensitive
print("cat" == "Cat")            # False
print("cat".lower() == "Cat".lower())  # True

# Locale-aware comparison
locale.setlocale(locale.LC_COLLATE, "de_DE.UTF-8")  # German
print(locale.strcoll("äpfel", "apfel"))  # May treat ä ~ ae

locale.setlocale(locale.LC_COLLATE, "sv_SE.UTF-8")  # Swedish
print(locale.strcoll("ä", "z"))  # ä sorted after z

# Natural sorting with numbers
files = ["file2", "file10", "file1"]
print(sorted(files))   # Lexicographic: ['file1', 'file10', 'file2']
files.sort(key=lambda x: [int(n) if n.isdigit() else n for n in re.split(r'(\d+)', x)])
print(files)           # Natural sort: ['file1', 'file2', 'file10']
```

Why it matters

Collation and locale rules affect almost every application that processes human text. Sorting names in a contact list, comparing file names, or filtering database queries all rely on comparison rules. Incorrect handling leads to confusing results for users, especially in multilingual contexts.

Natural sorting improves readability when working with numbered data. Database collations ensure consistency in queries and reports.

Exercises

1. Compare `"cat"` and `"Cat"` both case-sensitive and case-insensitive.
2. Explain why `"file10"` comes before `"file2"` in lexicographic order.
3. Show how converting both strings to lowercase helps with case-insensitive comparison.
4. Describe how `"ä"` is treated differently in German vs Swedish ordering.
5. Given a list of filenames (`file1`, `file2`, `file10`), sort them in natural order.
6. Explain how a database collation can make `"résumé"` equal to `"resume"`.
7. Show how case-insensitive and accent-insensitive comparisons change the equality of `"café"` and `"CAFE"`.
8. Create a rule set for sorting words that mix numbers and letters, like `"a2"`, `"a10"`, `"a3"`.
9. Demonstrate how incorrect collation could mis-sort a list of international names.
10. Design an algorithm for locale-aware string comparison that handles both case and accent differences.

3.3 L2 — Deep Comparisons

At the advanced level, comparing strings requires knowledge of Unicode, normalization, system-level optimizations, and specialized libraries. Equality and ordering are no longer trivial: they must handle multiple scripts, cultural rules, and performance constraints at scale.

Deep comparison of strings involves more than checking characters one by one. Unicode introduces multiple ways to represent the same text. Collation rules vary across locales. Large-scale systems need efficient comparison functions that work correctly with international text while remaining fast enough for billions of operations.

Deep Dive

Unicode Normalization

The same character can be represented in different ways. For example, `"é"` can be stored as a single code point (`U+00E9`) or as `'e'` plus a combining accent (`U+0065 U+0301`). Normalization ensures consistent representation. NFC (composed) and NFD (decomposed) are the most common forms.

Case Folding

Case-insensitive comparisons must use case folding, which is more complex than simply converting to lowercase. For instance, the German `"ß"` becomes `"ss"` when folded.

Collation with ICU

The International Components for Unicode (ICU) library provides robust comparison and sorting. It supports locale-sensitive rules, accent sensitivity, and custom collation. For example, `"résumé"` may equal `"resume"` in accent-insensitive mode.

System-Level Implementations

At low levels, comparison often reduces to optimized memory functions. Functions like `memcmp` compare bytes quickly, using CPU instructions that process multiple bytes at once. Short-circuit evaluation stops at the first mismatch. Cache alignment improves throughput when scanning long strings.

Security Concerns

Unicode confusables (characters that look alike, such as Latin `"a"` and Cyrillic `" "`) can fool comparison routines if normalization is not enforced. Attackers exploit this in phishing and injection attacks.

Challenge	Example	Solution
Multiple representations	`"é"` vs `"e\u0301"`	Normalize (NFC/NFD)
Case folding	`"Straße"` vs `"STRASSE"`	Unicode-aware case folding
Collation differences	`"ä"` in German vs Swedish	ICU locale rules
Confusables	`"a"` vs `" "`	Security checks, mapping

Worked Example (Python)

```
import unicodedata
import locale

# Unicode normalization
s1 = "café"          # composed
s2 = "cafe\u0301"    # decomposed
print("Raw equal:", s1 == s2)
print("NFC equal:", unicodedata.normalize("NFC", s1) == unicodedata.normalize("NFC", s2))
print("NFD equal:", unicodedata.normalize("NFD", s1) == unicodedata.normalize("NFD", s2))

# Case folding
print("ß".casefold() == "ss")   # True
```

```
# Locale-sensitive ordering
locale.setlocale(locale.LC_COLLATE, "fr_FR.UTF-8")  # French
words = ["école", "eclair", "étude"]
print(sorted(words, key=locale.strxfrm))  # Locale-aware sort

# Confusables
latin_a = "a"
cyrillic_a = " "  # visually similar
print("Equal?", latin_a == cyrillic_a)
print("Latin:", ord(latin_a), "Cyrillic:", ord(cyrillic_a))

# System-level demonstration (byte comparison)
b1 = b"apple"
b2 = b"apricot"
print("memcmp style:", (b1 > b2) - (b1 < b2))  # -1, 0, or 1
```

Why it matters

Deep string comparison underpins databases, search engines, and operating systems. Normalization ensures data consistency across storage and transfer. Case folding avoids incorrect matches in multilingual systems. ICU collation makes applications usable across cultures. Optimized byte comparisons keep performance acceptable at scale. Security requires detecting confusables to prevent spoofing. Without these techniques, global systems would be unreliable and vulnerable.

Exercises

1. Give an example where two visually identical strings are unequal because of different Unicode encodings.
2. Normalize two strings with accents and show they become equal.
3. Explain why `"ß"` must be compared to `"ss"` in case-insensitive matching.
4. Sort the words `"école"`, `"eclair"`, `"étude"` using French collation rules.
5. Show why `"a"` (Latin) and `" "` (Cyrillic) are dangerous in string comparison.
6. Describe how `memcmp` works at the system level to compare two byte sequences.
7. Explain how cache alignment improves performance when comparing long strings.
8. Demonstrate the difference between accent-sensitive and accent-insensitive comparison with `"café"` vs `"cafe"`.
9. Propose a method to detect and mitigate Unicode confusables in user input.
10. Design a benchmark to compare normalized vs non-normalized string comparison in a large dataset.

3.4 Patterns

3.4 L0 — Simple Search

Strings are often used to find and replace text. Beginners need to understand how to look for substrings, replace parts of text, and use simple wildcards. These operations are the first step toward more advanced pattern matching.

Overview/Definition

A search operation checks if a smaller string exists inside a larger one. If found, it returns the position or a confirmation. Replace operations swap one substring for another. Wildcards act as placeholders, matching characters in flexible ways. These basics allow programs to search, clean, and transform text.

Deep Dive

Finding Substrings

The simplest search asks: does `"dog"` appear inside `"hotdog"`? Most systems return the starting index of the match. If not found, a special value is returned.

Replacing Text

Replacement takes an old value and swaps it with a new one. `"the sky is blue"` with `"blue" → "red"` becomes `"the sky is red"`. Replacement is useful for cleaning or formatting text.

Wildcards (Intuitive View)

Wildcards are simple symbols that represent "any character." A `*` can mean "any sequence," and a `?` can mean "one character." Example: pattern `"ca*"` matches `"cat"`, `"car"`, and `"castle"`. Wildcards are simpler than full regular expressions but build the same intuition.

Operation	Example Input	Result
Contains	`"cat"` in `"concatenate"`	True
Find index	`"hello".find("lo")`	3
Replace	`"2025-09-10".replace("-", "/")`	`"2025/09/10"`
Wildcard	Pattern `"ca*"` vs `"castle"`	Match

Worked Example (Python)

```
# Searching substrings
sentence = "the quick brown fox"
print("quick" in sentence)       # True
print(sentence.find("fox"))      # 16
print(sentence.find("dog"))      # -1 (not found)

# Replacing text
date = "2025-09-10"
print(date.replace("-", "/"))    # 2025/09/10

# Simple wildcard matching (manual example)
def matches(pattern, text):
    if pattern == "*":  # match everything
        return True
    if pattern.endswith("*"):
        return text.startswith(pattern[:-1])
    if pattern.startswith("*"):
        return text.endswith(pattern[1:])
    return pattern == text

print(matches("ca*", "cat"))      # True
print(matches("ca*", "castle"))   # True
print(matches("*ing", "running")) # True
print(matches("*at", "cat"))      # True
```

Why it matters

Searching and replacing text is one of the most common tasks in programming. From finding a word in a document, to cleaning up user input, to formatting data for storage, these operations are essential. Wildcards introduce the idea of flexible matching, which prepares learners for regular expressions and more advanced pattern recognition.

Exercises

1. Check if the word `"sun"` appears in the sentence `"the sun sets"`.
2. Find the position of `"moon"` in `"the moon rises"`.
3. Replace `"morning"` with `"evening"` in `"good morning"`.
4. Show that searching for `"dog"` in `"hotdog"` succeeds.

5. Replace all dashes in `"2025-09-10"` with slashes.
6. Write a rule with `*` that matches both `"cat"` and `"castle"`.
7. Create a wildcard pattern that matches any word ending in `"ing"`.
8. Explain why a failed search must return a special value (not just `0`).
9. Demonstrate how searching for `"a"` in `"banana"` should return multiple matches.
10. Describe how wildcards can be extended into full regular expressions.

3.4 L1 — Regular Expressions in Practice

At the intermediate level, string pattern matching expands from simple wildcards to regular expressions (regex). Regex provides a concise and powerful way to describe patterns in text. It is widely used in programming for searching, extracting, and validating structured data.

A regular expression is a sequence of symbols that defines a search pattern. Unlike simple wildcards, regex can express detailed rules such as "find all numbers," "match words ending in `ing`," or "validate an email address." Regex is supported in almost every major programming language.

Deep Dive

Core Symbols

- `.` matches any single character.
- `*` means zero or more repetitions.
- `+` means one or more repetitions.
- `?` means zero or one occurrence.
- `[abc]` matches one character from the set `a`, `b`, or `c`.
- `[0-9]` matches any digit.
- `^` matches the beginning of a string; `$` matches the end.

Examples of Patterns

- `[0-9]+` matches sequences of digits like `"123"`.
- `\w+` matches words made of letters, digits, and underscores.
- `\w+@\w+\.\w+` matches simple email addresses.
- `^Hello` matches any string starting with `"Hello"`.

Greedy vs Lazy Matching

Regex quantifiers (`*`, `+`) are greedy by default: they try to match as much as possible. Adding `?` makes them lazy, matching the smallest possible part.

- Greedy: pattern `".*"` on `"abc"` → `"abc"`.
- Lazy: pattern `".*?"` on `"abc"` → `"a"`.

Practical Uses

Regex is used for:

- Extracting structured data from logs.
- Validating input like emails or phone numbers.
- Searching and replacing complex text patterns.
- Parsing configuration files or text protocols.

Worked Example (Python)

```
import re

text = "Order 123, item 456, email: alice@example.com"

# Match digits
numbers = re.findall(r"[0-9]+", text)
print("Numbers:", numbers)  # ['123', '456']

# Match words
words = re.findall(r"\w+", text)
print("Words:", words)

# Match email addresses
emails = re.findall(r"\w+@\w+\.\w+", text)
print("Emails:", emails)

# Anchors
print(bool(re.match(r"^Order", text)))   # True (starts with Order)
print(bool(re.search(r"com$", text)))   # True (ends with com)

# Greedy vs lazy
sample = "<tag>content</tag><tag>other</tag>"
greedy = re.findall(r"<tag>.*</tag>", sample)
```

```
lazy   = re.findall(r"<tag>.*?</tag>", sample)
print("Greedy:", greedy)  # ['<tag>content</tag><tag>other</tag>']
print("Lazy:", lazy)      # ['<tag>content</tag>', '<tag>other</tag>']
```

Why it matters

Regex provides a compact language for describing patterns in text. Without it, tasks like extracting emails, validating formats, or parsing logs would require many lines of manual code. Regex is also standardized across languages, so learning it once provides a universal tool. However, careless use can create unreadable code or performance problems, so disciplined practice is essential.

Exercises

1. Write a regex that matches any sequence of digits.
2. Match all words in `"The quick brown fox"`.
3. Find all substrings that start with `"a"` and end with `"e"`.
4. Match an email address of the form `"name@domain.com"`.
5. Match any string that begins with `"http://"` or `"https://"`.
6. Extract all numbers from `"Invoice: #123, #456, #789"`.
7. Demonstrate the difference between greedy and lazy matching with `"<tag>text</tag><tag>more</tag>"`.
8. Write a regex to validate phone numbers like `"123-456-7890"`.
9. Explain why using regex to parse complex formats (like HTML) can be unreliable.
10. Create a regex that matches words containing exactly three vowels.

3.4 L2 — Engines & Optimizations

Regular expressions are powerful, but at scale their performance and safety depend on the design of the engine. Advanced systems need regex engines that are predictable, fast, and safe against denial-of-service attacks. At this level, the focus is on internals, optimizations, and production libraries.

Regex engines are implementations of pattern matching. Some use backtracking, exploring many paths but risking exponential slowdowns. Others use deterministic finite automata (DFA) or non-deterministic automata (NFA), which guarantee linear runtime but reduce flexibility. Modern libraries combine these with JIT compilation or SIMD acceleration for speed. Production systems must choose the right engine for correctness and reliability.

Deep Dive

Engine Architectures

Backtracking engines (Perl, Python, PCRE) allow advanced features like backreferences but may run in exponential time on crafted inputs. DFA/NFA engines (RE2, Rust, grep) convert patterns to automata that run in linear time. JIT-based engines (Java, .NET) compile regex into machine code for faster execution.

Greedy vs Lazy at Engine Level

Quantifiers like `*` and `+` are greedy by default. Engines track multiple possibilities with stacks or state machines. Lazy quantifiers stop early, but the engine still evaluates alternatives internally.

Performance Optimizations

Compiled regex patterns are faster than compiling on every use. Anchors (`^`, `$`) and character classes (`[0-9]`) reduce search space. Avoiding pathological expressions like `(a+)+` prevents catastrophic backtracking. SIMD instructions scan text in chunks, speeding up low-level matching.

Advanced Libraries

- RE2 (Google): guarantees linear runtime, disallows constructs that cause exponential behavior.
- Hyperscan (Intel): supports multi-pattern regex with SIMD acceleration, used in intrusion detection.
- ICU: supports Unicode-aware regex and locale-sensitive operations.

System-Level Integration

Regex engines are embedded in compilers, text editors, search tools, and security systems. Multi-pattern matching is essential for scanning logs, detecting spam, and monitoring network traffic.

Security Concerns

Poorly written regex can create ReDoS (Regex Denial of Service). Attackers send inputs that trigger exponential backtracking, freezing systems. For example, pattern `(a+)+b` against `"aaaaaaa…"` takes excessive time. Safe libraries (RE2, Hyperscan) prevent this by design.

Engine Type	Strengths	Weaknesses
Backtracking	Full regex features, flexible	Risk of exponential runtime (ReDoS)
DFA/NFA	Linear runtime, predictable	No backreferences, fewer features
JIT Hybrid	Very fast, compiled to machine code	Still vulnerable to bad patterns

Worked Example (Python)

```
import re, time

# Backtracking danger: catastrophic regex
pattern = re.compile(r"(a+)+b")
text = "a" * 25 + "b"   # works
print("Match:", bool(pattern.match(text)))

# Problematic input: missing final 'b'
text_bad = "a" * 25     # exponential backtracking
start = time.time()
print("Bad match:", bool(pattern.match(text_bad)))
print("Time:", round(time.time() - start, 4), "seconds")

# Greedy vs lazy
html = "<tag>first</tag><tag>second</tag>"
greedy = re.findall(r"<tag>.*</tag>", html)
lazy   = re.findall(r"<tag>.*?</tag>", html)
print("Greedy:", greedy)
print("Lazy:", lazy)

# Precompiled pattern efficiency
words = "error warning info debug error warning".split()
p = re.compile(r"error|warning")
matches = [w for w in words if p.match(w)]
print("Matches:", matches)

# Unicode-aware regex
text = "café resume résumé"
utf8_words = re.findall(r"\w+", text)
print("UTF-8 words:", utf8_words)

# Simulated natural sorting with regex
files = ["file1", "file10", "file2"]
```

```
splitter = re.compile(r'(\d+)')
files.sort(key=lambda x: [int(n) if n.isdigit() else n for n in splitter.split(x)])
print("Natural sort:", files)
```

Why it matters

Regex is a double-edged sword. It compresses complex logic into concise expressions, but the wrong engine or pattern can cripple systems. Backtracking engines offer flexibility but can be exploited. DFA-based engines guarantee performance but restrict features. Production libraries like RE2 and Hyperscan provide safe, optimized solutions for real workloads. Security, speed, and predictability all depend on understanding regex internals.

Exercises

1. Explain the difference between greedy and lazy matching.
2. Show why `(a+)+b` can cause problems on inputs like `"aaaaaa"`.
3. Compare backtracking and DFA approaches to regex.
4. Write a regex to match numbers at the start of a string (`^123`).
5. Explain why precompiling a regex can improve performance.
6. Describe how RE2 avoids catastrophic backtracking.
7. Show why regex should not be used for parsing HTML.
8. Design a regex for matching IPv4 addresses and explain performance concerns.
9. Compare the runtime of regex matching with and without SIMD acceleration in theory.
10. Propose a strategy to defend against Regex Denial of Service in a production system.

3.5 Applications

3.5 L0 — Everyday Uses

Strings are used in almost every program. Beginners usually encounter them when printing messages, saving text to files, or checking simple inputs. These small applications show how strings connect code to real-world tasks.

Everyday string applications include formatting output, working with text files, validating simple input, and combining small pieces of data into readable results. These tasks look simple but form the foundation of real-world software, from command-line tools to web apps.

Deep Dive

Text Formatting

Programs must often display results with both words and numbers. Concatenation creates readable sentences: `"Hello, " + name`. Many languages also support templates: `"Hello, {name}"`.

File Reading Basics

A file is just text stored on disk. Reading one line returns a string. Splitting turns the line into words. Writing strings saves output. Example: reading `"Alice 25"` and splitting it into a name and an age.

Input Validation (Simple)

Not all inputs are valid. Checking that a string is not empty prevents errors. Minimum length ensures requirements are met (like passwords). Character checks confirm only letters or digits are entered.

Basic Data Handling

Strings often need simple processing: counting characters, trimming spaces, or extracting parts. `"John Doe"` can be split into `"John"` and `"Doe"`. Small pieces of text are often joined into a readable result.

Everyday Task	Example Input	Example Output
Greeting	`"Alice"`	`"Hello, Alice!"`
Trim spaces	`" hello "`	`"hello"`
Count characters	`"banana"`	`6`
Save and load message	`"Note: call Bob"`	Written and reloaded

Worked Example (Python)

```
# Formatting text
name = "Alice"
age = 25
print(f"Hello, {name}! You are {age} years old.")

# File basics (write then read)
```

```
with open("note.txt", "w") as f:
    f.write("Remember to buy milk")

with open("note.txt", "r") as f:
    content = f.read()
print("File content:", content)

# Input validation
user_input = "   hello123   "
cleaned = user_input.strip()
if cleaned and cleaned.isalnum():
    print("Valid input:", cleaned)
else:
    print("Invalid input")

# Data handling
full_name = "John Doe"
first, last = full_name.split()
print("First name:", first)
print("Last name:", last)

# Joining pieces
words = ["data", "structures", "are", "fun"]
sentence = " ".join(words)
print(sentence)
```

Why it matters

Everyday string handling bridges user interaction and program logic. Without formatting, programs would only output raw numbers. Without validation, programs would accept broken or unsafe input. Without file reading and writing, no program could save notes or logs. These basics show how strings connect software to people and data.

Exercises

1. Print `"Hello, NAME!"` using a variable for the name.
2. Count the number of characters in `"banana"`.
3. Save the string `"study algorithms"` to a file, then read it back.
4. Remove extra spaces from `"   world   "` and print the cleaned string.
5. Check if `"abc123"` contains only letters and digits.
6. Split `"Alice Johnson"` into first and last names.

7. Join `["red", "green", "blue"]` into `"red,green,blue"`.
8. Create a greeting that includes both a name and an age.
9. Validate that a password is at least 8 characters long.
10. Write a program that asks for a note, saves it to a file, and then reloads and prints it.

3.5 L1 — Engineering Contexts

As programs grow, string handling shifts from small tasks to structured data and system integration. Strings become the glue for configurations, logs, and filtering. These applications demand clarity, consistency, and efficiency.

Overview/Definition

Strings in engineering contexts are used to represent structured formats, record system activity, and enable searching and filtering. They provide a common medium for both humans and machines, balancing readability with machine-parsable structure.

Deep Dive

Configuration Formats

Configuration files define how systems run. Common formats like JSON, YAML, or INI are all string-based. Parsing these strings gives structured values. For example, `"timeout": 30` in JSON is a string representation of a setting, later converted into a number.

Logs and Monitoring

Systems record events as text logs. A log line might be `"2025-09-10 12:00:01 ERROR Connection failed"`. These strings are later parsed for monitoring and debugging. Consistency in string format is critical.

Searching and Filtering

Large systems must sift through text quickly. Searching log files for `"ERROR"` highlights problems. Filtering configuration values based on keywords is another common task. Case-insensitive matching or regex-based extraction makes this robust.

Data Exchange

APIs and network protocols often transmit structured text. JSON payloads are strings at the boundary between systems. Converting between string representation and structured data is a common responsibility of engineering code.

Context	Example String	Purpose
Configuration	`{"host":"localhost","port":8080}`	System settings
Log line	`2025-09-10 ERROR Disk full`	Monitoring & debugging
Search/filter	Find `"ERROR"` in a log file	Identify issues
API payload	`{"user":"alice","id":42}`	Data exchange between services

Worked Example (Python)

```python
import json
import re

# Configuration parsing (JSON as string)
config_str = '{"host":"localhost","port":8080}'
config = json.loads(config_str)
print("Host:", config["host"], "Port:", config["port"])

# Logging and filtering
logs = [
    "2025-09-10 12:00:01 INFO  Server started",
    "2025-09-10 12:01:22 ERROR Connection failed",
    "2025-09-10 12:02:45 WARN  Disk almost full"
]
errors = [line for line in logs if "ERROR" in line]
print("Error logs:", errors)

# Case-insensitive search
query = "warn"
matches = [line for line in logs if query.lower() in line.lower()]
print("Warnings:", matches)

# Extracting structured data from logs with regex
pattern = re.compile(r"(\d{4}-\d{2}-\d{2})\s+\S+\s+(ERROR|WARN|INFO)")
for log in logs:
    match = pattern.search(log)
```

```
    if match:
        date, level = match.groups()
        print(f"Date: {date}, Level: {level}")
```

Why it matters

Strings in configurations allow flexible system tuning without recompiling. Logs provide the primary record of system behavior and failures. Searching and filtering turn raw logs into actionable insights. APIs and protocols rely on structured strings to communicate. Mastery of these applications makes code more reliable and systems more transparent.

Exercises

1. Parse a JSON string that contains a username and print the value.
2. Find all log lines that contain `"ERROR"`.
3. Perform a case-insensitive search for `"warn"` in a list of log lines.
4. Extract the date from a log entry of the form `"YYYY-MM-DD ..."`.
5. Parse a configuration string to extract host and port values.
6. Write a filter that selects only `"INFO"` level log lines.
7. Convert a dictionary of settings into a JSON string.
8. Explain why consistent formatting in logs is important for monitoring systems.
9. Create a regex that extracts both the date and log level from a log entry.
10. Design a simple log query: given a list of log strings and a keyword, return all lines that match the keyword.

3.5 L2 — Large-Scale Systems

When systems handle massive amounts of text, strings are no longer just for messages, logs, or configs. They become the raw material of search engines, data compression, natural language processing, and security enforcement. At this scale, efficiency, correctness, and safety are critical.

Strings in large-scale systems must be stored compactly, searched efficiently, processed for meaning, and sanitized for safety. Specialized algorithms and libraries turn raw text into structured, searchable, and secure data.

Deep Dive

Search Engines

Full-text search systems use inverted indexes: instead of storing text linearly, they map each word to the documents it appears in. Searching `"cat"` means looking up `"cat"` in the index rather than scanning every file. Ranking algorithms (like TF-IDF or BM25) then score relevance.

Data Compression

Strings consume storage. Compression algorithms like Huffman coding, gzip, or dictionary-based schemes reduce size. For example, `"banana banana banana"` can be compressed by storing `"banana"` once and pointing to it three times. Compression reduces storage costs and speeds up network transfer.

Natural Language Processing (NLP)

Raw strings must be tokenized (split into words), normalized (case folding, stemming), and sometimes embedded into vectors. `"running"` $\rightarrow$ `"run"`. Preprocessing ensures machine learning models treat text consistently.

Security

Strings are also attack vectors. SQL injection, XSS, and command injection happen when untrusted strings are executed as code. Escaping or sanitizing input prevents this. Unicode tricks (confusables, hidden nulls) are used in phishing and bypass attacks. Secure systems enforce normalization and validation before processing.

Domain	String Role	Techniques
Search engine	Find text in billions of docs	Inverted index, ranking
Compression	Reduce text storage and transfer cost	Huffman, gzip, dictionary
NLP	Prepare text for analysis	Tokenization, stemming
Security	Prevent malicious string usage	Escaping, validation

Worked Example (Python)

```
import re
import zlib
from collections import defaultdict

# --- Search Engine Simulation ---
documents = {
    1: "the quick brown fox",
    2: "the lazy dog",
    3: "the fox jumped over the dog"
}

# Build inverted index
index = defaultdict(list)
for doc_id, text in documents.items():
    for word in text.split():
        index[word].append(doc_id)

print("Index for 'fox':", index["fox"])  # [1, 3]

# --- Compression Example ---
text = "banana banana banana"
compressed = zlib.compress(text.encode("utf-8"))
decompressed = zlib.decompress(compressed).decode("utf-8")
print("Original:", len(text), "bytes")
print("Compressed:", len(compressed), "bytes")
print("Decompressed:", decompressed)

# --- NLP Tokenization & Normalization ---
raw = "Running runs runner"
tokens = raw.lower().split()
stems = [re.sub(r"(ing|s|er)$", "", t) for t in tokens]  # naive stemming
print("Tokens:", tokens)
print("Stems:", stems)

# --- Security Example ---
unsafe_input = "'; DROP TABLE users; --"
safe_query = "SELECT * FROM accounts WHERE name = ?"
print("Unsafe input:", unsafe_input)
print("Safe query template:", safe_query)
```

Why it matters

Large-scale string handling powers web search, messaging apps, compression in files and networks, and text-based AI systems. Without inverted indexes, search engines would be unusably slow. Without compression, storage and bandwidth would be wasteful. Without normalization, machine learning models would misinterpret text. Without security checks, attackers could exploit untrusted strings to damage systems. Strings at scale are the backbone of modern computing.

Exercises

1. Explain how an inverted index speeds up searching compared to scanning all documents.
2. Show how compression reduces storage for repetitive strings like `"abc abc abc"`.
3. Tokenize `"Cats are running"` into lowercase words.
4. Normalize `"Running"` into its root form `"run"`.
5. Describe why `"café"` and `"cafe"` should be normalized before search.
6. Give an example of SQL injection using a string input.
7. Explain how escaping special characters prevents injection attacks.
8. Compare the storage savings between raw and compressed text for a long repeated string.
9. Propose how to detect Unicode confusables in user names.
10. Design a small experiment to measure search speed with and without an inverted index.

Deep Dive

3.1 Representation

- L0:
  - Immutability in everyday languages (intuitive).
  - Indexing and slicing basics.
  - Concatenation and repetition.
  - ASCII vs Unicode (conceptual).
- L1:
  - Immutability and its performance costs.
  - String interning and memory reuse.
  - Encoding differences: UTF-8, UTF-16, UTF-32.
  - Slicing behavior across languages (Python vs Java vs C++).
  - Efficient concatenation (`join`, builders, buffers).
- L2:

  - Language-specific internals: C `char*`, C++ `std::string` + SSO, Python PEP 393, Java `String`, Go slices, Rust `String`/`&str`.
  - OS-level considerations: heap vs stack, fragmentation, system call boundaries.
  - Hardware-level: cache alignment, SIMD optimization in libc.
  - Unicode normalization (NFC, NFD, case folding).
  - Security issues: confusables, null injection.
  - ICU, RE2, Hyperscan as production-grade libraries.

3.2 Operations

- L0:
  - Case conversion (upper, lower).
  - Trimming whitespace.
  - Basic search (`find`, `in`).
  - Splitting and joining.
  - Simple string formatting.
- L1:
  - Efficient searching (`startswith`, `endswith`).
  - Counting substrings and replacing.
  - Regular expression basics (`.`, `*`, `+`, `[]`).
  - Parsing structured text (CSV, JSON).
  - Performance pitfalls (naive concatenation, multiple scans).
- L2:
  - String search algorithms: Naive, KMP, Boyer–Moore.
  - Regex engine internals (backtracking vs DFA/NFA).
  - Tokenization in compilers (lexical analysis with automata).
  - Unicode-aware searching and case folding.
  - System boundary handling (sockets, files, encodings).
  - SIMD-accelerated substring search.

3.3 Comparison

- L0:
  - Equality checks (case-sensitive, whitespace).
  - Lexicographic ordering.
  - Sorting by dictionary order.

- L1:
  - Case-insensitive comparisons.
  - Locale-aware collation (German vs Swedish).
  - Natural sorting with numbers (`file2` < `file10`).
  - Database collations (CI/CS, AI/AS).
- L2:
  - Unicode normalization for equality.
  - Case folding vs lowercase.
  - ICU collation internals.
  - System-level optimizations (memcmp, short-circuit).
  - Cache-aware comparison for large text.
  - Detecting and handling Unicode confusables.

3.4 Patterns

- L0:
  - Simple substring search.
  - Replace operations.
  - Wildcards (`*`, `?`) as intuitive pattern matching.
- L1:
  - Regex syntax: quantifiers, anchors, character classes.
  - Greedy vs lazy matching.
  - Regex for input validation (emails, phones).
  - Extracting structured data from logs.
- L2:
  - Regex engine architectures: backtracking, DFA/NFA, hybrid JIT.
  - Performance optimizations (precompiled regex, anchors, SIMD).
  - Production libraries: RE2, Hyperscan, ICU.
  - System integration: compilers, search tools, IDS.
  - Security: catastrophic backtracking, ReDoS.

3.5 Applications

- L0:
  - Simple greetings and message formatting.
  - Reading and writing small text files.
  - Input validation (non-empty, length, alphanumeric).
  - Joining/extracting small strings.
- L1:
  - Configurations (INI, JSON, YAML).
  - Logs and monitoring pipelines.
  - Searching and filtering text data.
  - APIs and network payloads as strings.
- L2:
  - Search engines with inverted indexes.
  - Text compression (Huffman, gzip, dictionary).
  - Natural language preprocessing (tokenization, stemming).
  - Security concerns: injections, normalization, Unicode attacks.
  - High-scale performance engineering for string-heavy systems.

LAB

3.1 Representation

- LAB 1: String Encodings in Practice
  - Represent the same text (`"A "`) in UTF-8, UTF-16, and UTF-32.
  - Measure memory size in each encoding.
  - Show how slicing works differently in each.
- LAB 2: String Interning and Memory Efficiency
  - Create repeated strings with and without interning.
  - Measure memory usage.
  - Discuss practical use cases (symbol tables, compilers).
- LAB 3: Kernel-Level String Operations
  - Explore C `char*` strings with `strlen` and buffer overflows.
  - Compare to higher-level language safety (Python, Java).

3.2 Operations

- LAB 4: Concatenation Cost Benchmark
  - Compare loop concatenation (`+`) vs buffered approaches (`join`, builders).
  - Measure time for small, medium, and large inputs.
- LAB 5: Regex Basics Explorer
  - Implement small regex patterns (`[0-9]+`, `\w+`) across different languages.
  - Show differences in syntax and output.
- LAB 6: CSV Parsing Pitfalls
  - Try splitting `"apple,"banana,grape",cherry"` naively.
  - Show incorrect results.
  - Contrast with proper CSV parsers.

3.3 Comparison

- LAB 7: Locale-Aware Sorting
  - Sort a list with `"ä"`, `"z"`, `"apple"`.
  - Show German vs Swedish locale differences.
- LAB 8: Unicode Normalization Demo
  - Compare `"café"` vs `"cafe\u0301"`.
  - Show why normalization (NFC, NFD) matters for equality.
- LAB 9: Confusable Characters Security Check
  - Compare Latin `"a"` vs Cyrillic `" "`.
  - Build a detector for confusable Unicode characters.

3.4 Patterns

- LAB 10: Substring Search Algorithms
  - Implement Naive, KMP, and Boyer–Moore search.
  - Benchmark them on long repetitive text.
- LAB 11: Regex Engine Catastrophe
  - Use `(a+)+b` on long input without a `b`.

  - Measure how exponential backtracking freezes the engine.
  - Contrast with RE2 (linear-time safe).

- LAB 12: Greedy vs Lazy Matching

  - Run regex `"<tag>.*</tag>"` vs `"<tag>.*?</tag>"` on HTML-like text.
  - Show difference between greedy and lazy quantifiers.

3.5 Applications

- LAB 13: Build a Mini Inverted Index

  - Take a small set of documents.
  - Build an index mapping words to documents.
  - Query the index for `"fox"`, `"dog"`, etc.

- LAB 14: Compression of Repetitive Strings

  - Implement a simple dictionary compressor.
  - Compare original vs compressed sizes for `"banana banana banana"`.

- LAB 15: Injection Attack Simulation

  - Build a simple query string with unsanitized input.
  - Show how malicious input (`"'; DROP TABLE users; --"`) alters behavior.
  - Fix it with escaping or parameter binding.

LAB 1: String Encodings in Practice

Goal

Understand how different encodings (UTF-8, UTF-16, UTF-32) represent the same text. Learn how memory size changes depending on encoding, and why this matters for storage and transmission.

Setup

You can use any language that provides access to string encodings (Python, Go, C, Java). Python is a good choice because its `encode()` method supports multiple encodings.

Text sample:

`"A "`

This combines a simple ASCII character (`A`) with a multi-byte emoji ( ).

Step-by-Step

1. Represent the string in memory
    - Create the string `"A "`.
    - Print its length in characters.
    - Print its length in bytes when encoded in UTF-8, UTF-16, and UTF-32.
2. Inspect the raw byte sequences
    - Encode the string in each format.
    - Print the resulting byte arrays.
    - Write them in hexadecimal for clarity.
3. Compare differences
    - Notice that UTF-8 is variable-length (1 byte for `A`, 4 bytes for).
    - UTF-16 uses 2 bytes for `A`, 4 bytes for (surrogate pair).
    - UTF-32 uses 4 bytes for both.
4. Reflect on trade-offs
    - UTF-8 is compact for ASCII-heavy text (English).
    - UTF-16 is a balance for languages with many non-ASCII characters.
    - UTF-32 is simple but wastes space.

Example (Python)

```python
text = "A "
print("Characters:", len(text))

# UTF-8
utf8_bytes = text.encode("utf-8")
print("UTF-8 length:", len(utf8_bytes), "bytes")
print("UTF-8 bytes:", utf8_bytes.hex())

# UTF-16
utf16_bytes = text.encode("utf-16")
print("UTF-16 length:", len(utf16_bytes), "bytes")
print("UTF-16 bytes:", utf16_bytes.hex())

# UTF-32
utf32_bytes = text.encode("utf-32")
print("UTF-32 length:", len(utf32_bytes), "bytes")
print("UTF-32 bytes:", utf32_bytes.hex())
```

Expected output (approximate, may differ by platform endianness):

```
Characters: 2
UTF-8 length: 5 bytes
UTF-8 bytes: 41f09f988a
UTF-16 length: 6 bytes
UTF-16 bytes: fffe41003dd89a
UTF-32 length: 8 bytes
UTF-32 bytes: fffe0000410001f60a
```

Expected Results

Encoding	Bytes for `"A"`	Bytes for `" "`	Total Bytes
UTF-8	1	4	5
UTF-16	2	4 (surrogate)	6
UTF-32	4	4	8

Why it matters

- Encoding affects storage cost and speed.
- A file with millions of ASCII characters will be 2× bigger in UTF-16 than UTF-8.

- Emojis or non-Latin scripts can make UTF-8 larger than UTF-16.
- Choosing the wrong encoding can waste storage, slow down transmission, or even corrupt data if misinterpreted.

Exercises

1. Encode the string `"Hello"` in UTF-8, UTF-16, UTF-32. Compare sizes.
2. Repeat with `"  "` (Japanese). Which encoding is most efficient?
3. Inspect the raw bytes of `" "` and explain why it requires surrogate pairs in UTF-16.
4. Write a program that takes any string and reports its size in multiple encodings.
5. Explain why UTF-32 is rarely used for storage despite its simplicity.

LAB 2: String Interning and Memory Efficiency

Goal

Understand how string interning reduces memory usage by reusing identical string objects. Learn when interning happens automatically, when it must be explicit, and why it matters in systems with many repeated strings.

Setup

Use a language that supports string interning (Python, Java, C#, Go with symbol tables, etc.). We'll use Python because it demonstrates both automatic interning (for short literals) and manual interning (`sys.intern()`).

Test dataset: a large list of repeated keys (e.g., `"user123"`, `"error"`, `"warning"`).

Step-by-Step

1. Automatic Interning of Literals

    - Define two identical string literals.
    - Compare them with `==` (value equality) and `is` (identity).
    - Observe that they may point to the same object in memory.

2. Non-Interned Strings

    - Build identical strings at runtime (e.g., `"ab" + "cd"`).
    - Check identity with `is`.
    - They may not be the same object, even if values match.

3. Explicit Interning
    - Use `sys.intern()` on repeated dynamic strings.
    - Compare memory addresses before and after.
4. Memory Benchmark
    - Create a list of repeated values (e.g., 100k `"error"` strings).
    - Measure memory usage with and without interning.
5. Discussion
    - Interning saves memory by keeping one copy of repeated values.
    - It speeds up dictionary lookups (symbols, keywords, tokens).
    - But it increases memory if used on unique or rarely repeated strings.

Example (Python)

```
import sys

# 1. Automatic interning
a = "hello"
b = "hello"
print("a == b:", a == b)   # True (values equal)
print("a is b:", a is b)   # True (same object due to interning)

# 2. Non-interned strings
x = "".join(["he", "llo"])
y = "".join(["he", "llo"])
print("x == y:", x == y)   # True (values equal)
print("x is y:", x is y)   # False (different objects)

# 3. Manual interning
x_interned = sys.intern(x)
y_interned = sys.intern(y)
print("x_interned is y_interned:", x_interned is y_interned)  # True

# 4. Memory efficiency demo
words = ["error"] * 100000
unique_words = [sys.intern("error") for _ in range(100000)]
print("Normal list length:", len(words))
print("Interned list length:", len(unique_words))
```

```
print("Normal memory ids (first 3):", [id(w) for w in words[:3]])
print("Interned memory ids (first 3):", [id(w) for w in unique_words[:3]])
```

Expected Results

- String literals often share the same memory address.
- Dynamically built identical strings are separate objects unless interned.
- Interned strings reuse the same memory, reducing duplication.

Case	Identity (`is`)	Memory effect
`"hello"` vs `"hello"`	True	Interned automatically.
`"".join(["he","llo"])`	False	Different objects in memory.
`sys.intern()` applied	True	Both point to one shared object.

Why it matters

- Compilers and interpreters use interning for keywords, identifiers, and symbols.
- Databases and search engines save memory when handling millions of repeated strings.
- Performance improves because identity comparison (`is`) is faster than value comparison.
- Overuse of interning on unique strings wastes memory and CPU cycles.

Exercises

1. Create two identical literals and prove they point to the same memory location.
2. Build two identical strings dynamically and show they are different objects.
3. Apply interning to the dynamic strings and prove they now share memory.
4. Create 10,000 repeated `"apple"` strings with and without interning. Compare identity checks.
5. Explain why interning improves dictionary lookups.
6. Show an example where interning increases memory use (hint: many unique strings).
7. Compare interned vs non-interned strings in terms of lookup speed in a dictionary of 100k keys.
8. Describe how interning is used in language runtimes (e.g., symbol tables in compilers).
9. Propose a scenario in large-scale systems (like log processing) where interning saves significant memory.
10. Explain why interning must be combined with garbage collection to avoid memory leaks.

LAB 3: Kernel-Level String Operations

Goal

Explore how strings are represented at the C level using `char*`. Learn how null terminators (`'\0'`) define string boundaries, how buffer overflows occur, and why higher-level languages add safety.

Setup

This lab uses C (or C-like pseudocode). It can be run with `gcc` or `clang`. We'll demonstrate:

1. How C stores strings in memory.
2. What happens when the null terminator is missing.
3. Why functions like `strcpy` can cause buffer overflows.

Step-by-Step

1. String Representation in C
    - Create a C string with `char s[] = "hello";`.
    - Print its characters in a loop.
    - Show that memory includes `'\0'` at the end.
2. Effect of Null Terminator
    - Create a buffer without `'\0'`.
    - Pass it to `printf("%s")`.
    - Observe how the function reads past the intended end, printing garbage or crashing.
3. Buffer Overflow Risk
    - Create a small buffer (e.g., 5 bytes).
    - Copy a larger string into it with `strcpy`.
    - Show that memory beyond the buffer is overwritten.
4. Safer Alternatives
    - Replace `strcpy` with `strncpy` or `snprintf`.
    - Show how they limit copies and prevent overflows.

Example (C)

```c
#include <stdio.h>
#include <string.h>

int main() {
    // 1. String representation
    char s[] = "hello";
    printf("String: %s\n", s);
    for (int i = 0; i < sizeof(s); i++) {
        printf("Byte %d: %d\n", i, s[i]);
    }

    // 2. Missing null terminator
    char bad[5] = {'h','e','l','l','o'}; // no '\0'
    printf("Bad string (no null): %s\n", bad); // prints garbage

    // 3. Buffer overflow
    char small[5];
    strcpy(small, "overflow!"); // too long, unsafe
    printf("Overflowed string: %s\n", small);

    // 4. Safe alternative
    char safe[5];
    strncpy(safe, "safe", sizeof(safe) - 1);
    safe[sizeof(safe) - 1] = '\0'; // ensure null terminator
    printf("Safe string: %s\n", safe);

    return 0;
}
```

Expected Results

1. The string `"hello"` occupies 6 bytes: 5 characters + 1 null terminator.
2. Without `'\0'`, `printf` continues reading memory until it finds a random null.
3. `strcpy` into a small buffer overwrites adjacent memory, causing corruption or crashes.
4. `strncpy` keeps the buffer safe by limiting copied characters.

Case	Behavior
With `'\0'`	Prints correctly.

Case	Behavior
Without `'\0'`	Prints garbage or crashes.
Overflow with `strcpy`	Memory corruption.
Safe with `strncpy`	Correct, bounded copy.

Why it matters

- C-level strings are unsafe by default. The programmer must manage termination and bounds.
- Buffer overflows are a leading cause of security vulnerabilities.
- Kernel code, device drivers, and embedded systems often rely on raw C strings.
- Higher-level languages (Python, Java, Go, Rust) wrap strings in safe abstractions, preventing these bugs.

Exercises

1. Create a C string and print its bytes, showing the null terminator.
2. Remove the null terminator from a string and observe the output.
3. Copy `"abcdefgh"` into a buffer of size 5 and note the result.
4. Rewrite the unsafe copy using `strncpy` or `snprintf`.
5. Explain why `strncpy` may leave out the null terminator if not handled carefully.
6. Investigate what happens if you forget to set `safe[sizeof(safe)-1] = '\0'`.
7. Compare memory usage of `"hello"` vs `"hello world"`.
8. Explain why C strings make functions like `strlen` O(n) instead of O(1).
9. Show how buffer overflows can alter unrelated variables in memory.
10. Discuss how Rust or Go prevents these issues by design.

LAB 4: Concatenation Cost Benchmark

Goal

Measure the cost of string concatenation when building large strings. Understand why naive approaches (`+` in loops) are inefficient, and how buffered approaches (`join`, builders, buffers) improve performance.

Setup

This lab can be done in Python, Java, Go, or C++. Each has different idioms:

- Python → `+` vs `"".join(list)`
- Java → `+` vs `StringBuilder`
- Go → `+=` vs `strings.Builder`
- C++ → `+` vs `std::ostringstream`

We'll use Python as the demonstration language.

Step-by-Step

1. Naive Concatenation in Loop
    - Start with an empty string.
    - Append characters one by one with `+`.
    - Measure execution time.
2. Buffered Concatenation with List + Join
    - Store characters in a list.
    - Use `"".join(list)` to build the final string once.
    - Measure execution time.
3. Compare Results
    - Show how loop concatenation scales poorly ($O(n^2)$).
    - Show how buffered concatenation scales linearly ($O(n)$).
4. Experiment with Sizes
    - Run benchmarks for N = 10^3, 10 , 10 characters.
    - Record execution times.
5. Discussion
    - Naive concatenation reallocates and copies strings repeatedly.
    - Buffered methods allocate memory once and fill it efficiently.
    - Similar patterns exist across languages.

Example (Python)

```python
import time

def naive_concat(n):
    s = ""
    for i in range(n):
        s += "x"
    return s

def join_concat(n):
    parts = []
    for i in range(n):
        parts.append("x")
    return "".join(parts)

for N in [1000, 10000, 50000]:
    start = time.time()
    naive_concat(N)
    print(f"Naive concat N={N}: {round(time.time() - start, 4)}s")

    start = time.time()
    join_concat(N)
    print(f"Join concat N={N}: {round(time.time() - start, 4)}s")
    print("---")
```

Expected Results

- For small N (1000), both methods are similar.
- For larger N (50k+), naive concatenation slows dramatically.

N	Naive + (s)	Join (s)
1,000	~0.002	~0.001
10,000	~0.15	~0.01
50,000	~3.0+	~0.05

Why it matters

- Text-heavy applications (logs, report generation, data pipelines) must build large strings efficiently.
- Using the wrong method can multiply runtime by 100× or more.

- High-level languages hide memory allocation, but understanding efficiency prevents hidden bottlenecks.
- Systems languages (C++, Rust, Go) provide explicit builders to avoid repeated allocations.

Exercises

1. Benchmark string concatenation with `+` for N = 1000, 10,000, and 100,000.
2. Benchmark buffered concatenation (`join`, `StringBuilder`, or equivalent).
3. Plot execution time against N for both methods.
4. Explain why naive concatenation is $O(n^2)$.
5. Demonstrate that small strings ($N < 100$) show no noticeable difference.
6. Compare Python `join` with Go `strings.Builder`.
7. Create a function that takes a list of words and joins them with spaces efficiently.
8. Show how repeated concatenation can impact log processing in a simulated workload.
9. Extend the benchmark to include Unicode characters like `" "`.
10. Propose an optimization strategy for building multi-megabyte strings in a web server.

LAB 5: Regex Basics Explorer

Goal

Learn how regular expressions (regex) describe text patterns. Practice matching digits, words, and simple patterns across strings. Compare regex behavior across different languages, noting syntax similarities and differences.

Setup

Regex exists in almost every modern language.

- Python → `re` module
- Java → `Pattern` / `Matcher`
- Go → `regexp` package
- C++ → `<regex>`
- JavaScript → `/pattern/` literals

We'll demonstrate in Python for clarity, but the concepts transfer directly.

Step-by-Step

1. Basic Digit Matching
    - Use `[0-9]+` to match sequences of digits.
    - Extract numbers from a sentence.
2. Word Matching
    - Use `\w+` to capture words (letters, digits, underscores).
    - Show how it splits a sentence into tokens.
3. Anchors
    - Use `^pattern` to match the start of a string.
    - Use `pattern$` to match the end.
4. Character Classes
    - `[aeiou]` matches vowels.
    - `[A-Z]` matches uppercase letters.
5. Practical Example: Email
    - Use `\w+@\w+\.\w+` to match simple emails.
    - Note limitations (not RFC-complete, but useful).

Example (Python)

```
import re

text = "User Alice has email alice@example.com and id 12345."

# 1. Digits
digits = re.findall(r"[0-9]+", text)
print("Digits:", digits)  # ['12345']

# 2. Words
words = re.findall(r"\w+", text)
print("Words:", words)   # ['User', 'Alice', 'has', ...]

# 3. Anchors
print(bool(re.match(r"User", text)))   # True (starts with "User")
```

```
print(bool(re.search(r"12345$", text))) # True (ends with "12345.")

# 4. Character classes
vowels = re.findall(r"[aeiou]", text)
print("Vowels:", vowels)

# 5. Email pattern
emails = re.findall(r"\w+@\w+\.\w+", text)
print("Emails:", emails)
```

Expected Results

- `[0-9]+` extracts `"12345"`.
- `\w+` splits the sentence into words.
- `^User` matches the beginning.
- `[aeiou]` finds vowels.
- `\w+@\w+\.\w+` extracts `"alice@example.com"`.

Pattern	Meaning	Example Match
`[0-9]+`	One or more digits	`"12345"`
`\w+`	Word characters	`"Alice"`
`^User`	Must start with `"User"`	`"User Alice..."`
`[aeiou]`	Any single vowel	`"a", "e", "i"`
`\w+@\w+\.\w+`	Simple email pattern	`"alice@example.com"`

Why it matters

Regex is a universal tool for working with text: extracting data, validating input, or parsing logs. It is concise and expressive, replacing dozens of manual string operations. Learning the basics of regex builds the foundation for advanced pattern matching and text processing in any language.

Exercises

1. Write a regex to match all digits in `"room 101 floor 5"`.
2. Extract all words from `"The quick brown fox"`.
3. Match any string starting with `"Hello"`.
4. Match all vowels in `"banana"`.
5. Extract all email addresses from `"bob@mail.com, alice@test.org"`.

6. Write a regex that matches only uppercase words.
7. Find all three-letter words in `"the cat sat on the mat"`.
8. Show how `^` and `$` can be used to validate a string that must start and end with digits.
9. Explain why `\w+@\w+\.\w+` is not sufficient for real email validation.
10. Write a regex that finds all words ending in `"ing"` from a sentence.

LAB 6: CSV Parsing Pitfalls

Goal

Show why naive string splitting fails on CSV data. Demonstrate common pitfalls like quoted fields and embedded commas. Learn why dedicated CSV parsers are needed for reliability.

Setup

CSV (Comma-Separated Values) looks simple but has rules:

- Fields can be quoted with `"`
- Quoted fields can contain commas
- Quotes inside fields are escaped as `""`

We'll use Python, since it has both naive string methods and a built-in `csv` module.

Step-by-Step

1. Naive Split Works Sometimes
    - Take `"apple,banana,cherry"`.
    - Split by `,`.
    - Works fine.
2. Problem: Quoted Fields
    - Take `"apple,"banana,grape",cherry"`.
    - Split by `,`.
    - See how `"banana,grape"` is broken incorrectly.
3. Problem: Escaped Quotes
    - Take `"apple,"he said ""hi""",cherry"`.
    - Naive split breaks the field and does not handle quotes.

4. Correct Parsing with CSV Module
    - Use `csv.reader` to handle quotes and escapes.
    - Show correct results.
5. Discussion
    - Naive string splitting fails whenever quotes or embedded commas exist.
    - Correct parsers follow RFC 4180 rules.
    - Production code must use libraries, not ad hoc splitting.

Example (Python)

```
import csv

# 1. Naive split (works)
line1 = "apple,banana,cherry"
print("Naive split:", line1.split(","))  # ['apple', 'banana', 'cherry']

# 2. Quoted fields with commas
line2 = 'apple,"banana,grape",cherry'
print("Naive split:", line2.split(","))  # ['apple', '"banana', 'grape"', 'cherry']

# 3. Quoted fields with escaped quotes
line3 = 'apple,"he said ""hi""",cherry'
print("Naive split:", line3.split(","))  # ['apple', '"he said ""hi"""', 'cherry']

# 4. Correct parsing
for line in [line1, line2, line3]:
    parsed = next(csv.reader([line]))
    print("CSV parser:", parsed)
```

Expected Results

- Naive split fails on quoted fields.
- CSV parser handles commas and quotes correctly.

Input String	Naive Split	CSV Parser
`apple,banana,cherry`	`['apple','banana','cherry']`	`['apple','banana','cherry']`
`apple,"banana,grape",cherry`	`['apple','"banana','grape"','cherry']`	`['apple','banana,grape','cherry']`

Input String	Naive Split	CSV Parser
`apple,"he said ""hi""",cherry`	`['apple','"he said ""hi"""','cherry']`	`['apple','he said "hi"','cherry']`

Why it matters

CSV looks simple but is deceptively tricky. Many production bugs come from naive parsing that breaks when encountering quotes or embedded commas. Using proper libraries ensures reliability across systems. Understanding these pitfalls prevents data corruption in analytics, ETL pipelines, and configuration parsing.

Exercises

1. Write code that naively splits `"a,b,c"` and verify the output.
2. Split `"a,"b,c",d"` naively and explain what goes wrong.
3. Parse the same line using a CSV library and compare results.
4. Show how `"a,"he said ""ok""",c"` is incorrectly parsed by naive splitting.
5. Write a regex that attempts to parse CSV and explain its limits.
6. Compare parsing speed of naive split vs `csv.reader`.
7. Explain why using regex for full CSV parsing is impractical.
8. Show how a CSV parser handles an empty field (`"a,,c"`).
9. Create a CSV with newline characters inside quotes and parse it correctly.
10. Describe why standards like RFC 4180 exist for CSV and what problems they solve.

LAB 7: Locale-Aware Sorting

Goal

Understand how different locales affect string sorting. See why `"ä"` sorts differently in German vs Swedish, and how case sensitivity influences order. Learn why collation rules matter in real-world systems like databases and search engines.

Setup

Use a language that supports locale-aware collation:

- Python → `locale` module
- Java → `Collator`

- C++ → `<locale>` with `collate`
- Databases → `COLLATE` clauses

We'll demonstrate with Python.

Step-by-Step

1. Default Sorting (ASCII/Unicode order)
    - Sort a list of words: `["apple", "Banana", "äpfel", "zebra"]`.
    - Observe that uppercase letters and special characters do not sort as expected for dictionary order.
2. German Collation
    - Use `"de_DE.UTF-8"` locale.
    - In German, `"ä"` is often treated like `"ae"`.
    - Sort the same list and observe differences.
3. Swedish Collation
    - Use `"sv_SE.UTF-8"` locale.
    - In Swedish, `"ä"` is a separate letter that comes after `"z"`.
    - Compare results with German sorting.
4. Case Sensitivity
    - Compare `"apple"` vs `"Apple"`.
    - Show how some locales ignore case, others do not.
5. Discussion
    - Locale defines sorting rules for a culture or language.
    - Incorrect collation leads to confusing search results or misordered lists.
    - Databases use collations to enforce consistency.

Example (Python)

```python
import locale

words = ["apple", "Banana", "äpfel", "zebra"]

# 1. Default sorting
print("Default:", sorted(words))

# 2. German collation
locale.setlocale(locale.LC_COLLATE, "de_DE.UTF-8")
print("German:", sorted(words, key=locale.strxfrm))

# 3. Swedish collation
locale.setlocale(locale.LC_COLLATE, "sv_SE.UTF-8")
print("Swedish:", sorted(words, key=locale.strxfrm))

# 4. Case-insensitive comparison (normalize before sort)
case_insensitive = sorted(words, key=lambda w: w.lower())
print("Case-insensitive:", case_insensitive)
```

Expected Results

Locale	Sorted Output
Default (Unicode)	`['Banana', 'apple', 'zebra', 'äpfel']`
German (de_DE)	`['apple', 'äpfel', 'Banana', 'zebra']`
Swedish (sv_SE)	`['apple', 'Banana', 'zebra', 'äpfel']`
Case-insensitive	`['apple', 'äpfel', 'Banana', 'zebra']`

Why it matters

- Multilingual systems must handle cultural sorting correctly.
- `"ä"` is `"ae"` in German, but a distinct letter after `"z"` in Swedish.
- Case sensitivity can confuse users if `"Apple"` and `"apple"` appear far apart.
- Databases and search engines must define collation rules to ensure consistency.

Exercises

1. Sort `["Zoo", "apple", "Äpfel"]` in default order.
2. Sort the same list in German locale.

3. Sort the same list in Swedish locale.
4. Explain why results differ between German and Swedish.
5. Write code that performs case-insensitive sorting.
6. Compare how `"Résumé"` and `"resume"` sort under accent-sensitive vs accent-insensitive collation.
7. Demonstrate natural sorting for file names: `["file1", "file10", "file2"]`.
8. Explain how incorrect collation could break alphabetical order in a contact list.
9. Research and list collations available in your system/database.
10. Propose a default collation strategy for a global web application.

LAB 8: Unicode Normalization Demo

Goal

Show how visually identical strings can have different underlying Unicode encodings. Learn to use normalization (NFC, NFD) to make comparisons reliable across systems.

Setup

Unicode allows multiple representations of the same text:

- Precomposed characters (`é` as U+00E9)
- Decomposed sequences (`e` + U+0301 combining accent)

We'll use Python's `unicodedata` module to demonstrate.

Step-by-Step

1. Create Equivalent Strings
    - Define `s1 = "café"` (precomposed).
    - Define `s2 = "cafe\u0301"` (decomposed).
    - Print both — they look the same.
2. Compare Without Normalization
    - Check `s1 == s2`.
    - Result is `False`, even though they appear identical.
3. Normalize Strings
    - Apply NFC (Normalization Form Composed).

- Apply NFD (Normalization Form Decomposed).
- Compare results again.

4. Check Lengths and Code Points

   - Compare lengths of `s1` and `s2`.
   - Print their Unicode code points.
   - Show how precomposed has 1 code point for `é`, decomposed has 2.

5. Discussion

   - Normalization ensures consistent storage and comparison.
   - Databases and search engines often normalize text before indexing.
   - Without it, identical-looking words may be treated as different.

Example (Python)

```
import unicodedata

# 1. Create equivalent strings
s1 = "café"           # precomposed
s2 = "cafe\u0301"     # decomposed

print("s1:", s1)
print("s2:", s2)
print("Equal raw?:", s1 == s2)

# 2. Normalize
nfc1 = unicodedata.normalize("NFC", s1)
nfc2 = unicodedata.normalize("NFC", s2)
print("Equal NFC?:", nfc1 == nfc2)

nfd1 = unicodedata.normalize("NFD", s1)
nfd2 = unicodedata.normalize("NFD", s2)
print("Equal NFD?:", nfd1 == nfd2)

# 3. Inspect lengths and code points
print("Length s1:", len(s1))
print("Length s2:", len(s2))

print("Code points s1:", [hex(ord(c)) for c in s1])
print("Code points s2:", [hex(ord(c)) for c in s2])
```

Expected Results

- `s1 == s2` → False (different underlying encoding).
- After normalization (NFC or NFD), they compare equal.
- Lengths differ (`len(s1) == 4`, `len(s2) == 5`).
- Code points differ:
  - `s1`: `[0x63, 0x61, 0x66, 0xe9]`
  - `s2`: `[0x63, 0x61, 0x66, 0x65, 0x301]`

String	Visible Form	Encoding Style	Length	Code Points
s1	café	Precomposed NFC	4	c a f é (U+00E9)
s2	café	Decomposed NFD	5	c a f e + ´ (U+0301)

Why it matters

- Identical-looking text can fail equality checks if not normalized.
- File systems, databases, and APIs may use different normalization forms.
- Search engines normalize text to avoid duplicate results.
- Security-sensitive systems must normalize input to avoid spoofing or bypass attacks.

Exercises

1. Compare `"résumé"` written with precomposed vs decomposed accents.
2. Show how string lengths differ between NFC and NFD forms.
3. Write code to normalize a list of user inputs before storing them in a database.
4. Explain why `"é"` (U+00E9) is not equal to `"e" + U+0301`.
5. Demonstrate normalization on Japanese text (hiragana + diacritics).
6. Show how failing to normalize could cause duplicate entries in a dictionary.
7. Investigate what normalization form macOS uses by default for filenames.
8. Explain why normalization is critical in international domain names.
9. Create a function that reports whether two strings are canonically equivalent.
10. Design a normalization step for a search engine pipeline.

LAB 9: Confusable Characters Security Check

Goal

Demonstrate how visually identical characters can come from different scripts (Latin, Cyrillic, Greek). Show how this can fool string comparisons, and implement a basic detector for such confusables.

Setup

Attackers exploit Unicode confusables (e.g., Latin `"a"` vs Cyrillic `" "`) to trick users or bypass validation. This is called a homoglyph attack. Examples:

- `"paypal.com"` vs `"  l.com"` (Cyrillic and).
- `"admin"` vs `" dmin"` (Cyrillic).

We'll use Python for demonstration (`unicodedata` module).

Step-by-Step

1. Create Confusable Strings
    - Define `latin_a = "a"` and `cyrillic_a = " "` (U+0430).
    - Print them side by side.
    - Compare with `==` to show they are different.
2. Inspect Code Points
    - Use `ord()` to print Unicode code points.
    - Show the numeric difference despite identical appearance.
3. Confusable Detection
    - Write a function that checks if characters belong to different Unicode blocks.
    - Flag potential confusables.
4. Practical Example: Fake Domain
    - Construct `"  l.com"` (Cyrillic letters).
    - Compare it to `"paypal.com"`.
    - Show why naive string equality misses the difference.
5. Discussion

- Homoglyph attacks target login systems, domains, and usernames.
- Defenses:
  - Normalize + restrict allowed scripts.
  - Use libraries like Unicode confusables.txt.
  - Visual warnings in browsers.

Example (Python)

```
import unicodedata

# 1. Confusable characters
latin_a = "a"
cyrillic_a = " "  # U+0430
print("Latin a:", latin_a, "Cyrillic a:", cyrillic_a)
print("Equal?:", latin_a == cyrillic_a)

# 2. Inspect code points
print("Latin a code point:", hex(ord(latin_a)))
print("Cyrillic a code point:", hex(ord(cyrillic_a)))

# 3. Simple detector
def detect_confusables(text):
    scripts = {}
    for ch in text:
        name = unicodedata.name(ch, "")
        if "CYRILLIC" in name:
            scripts[ch] = "Cyrillic"
        elif "LATIN" in name:
            scripts[ch] = "Latin"
        elif "GREEK" in name:
            scripts[ch] = "Greek"
    return scripts

print("Detection:", detect_confusables("   l"))

# 4. Fake domain example
real = "paypal.com"
fake = "   l.com"  # with Cyrillic ' ' and ' '
print("Real domain:", real)
print("Fake domain:", fake)
```

```
print("Equal?:", real == fake)
print("Fake detection:", detect_confusables(fake))
```

Expected Results

- "a" vs " " look the same but are unequal.
- Code points differ:
  - Latin a → U+0061
  - Cyrillic → U+0430
- Detection function flags script differences.
- "paypal.com" " l.com", but naive eyes might miss it.

Character	Visual	Unicode	Script
a	a	U+0061	Latin
	a	U+0430	Cyrillic

Why it matters

- Security systems that ignore Unicode tricks are vulnerable to phishing and spoofing.
- Fake domains, usernames, or commands can bypass filters.
- Normalization does not fix confusables — script restriction or mapping is required.
- Modern browsers warn users about domains mixing scripts.

Exercises

1. Compare "p ypal" (with Cyrillic) to "paypal".
2. Print Unicode code points of "o" vs Cyrillic " ".
3. Extend the detect_confusables function to flag Greek letters too.
4. Create a fake username using mixed Latin and Cyrillic.
5. Show why normalization (NFC/NFD) does not fix confusables.
6. Research Unicode confusables.txt and find at least 5 common homoglyphs.
7. Propose a rule to allow only one script (Latin or Cyrillic) per string.
8. Write a program to detect mixed-script domains.
9. Explain how browsers handle homoglyph domains.
10. Design a security policy for usernames to prevent confusable attacks.

LAB 10: Substring Search Algorithms

Goal

Compare different substring search algorithms — Naive, Knuth–Morris–Pratt (KMP), and Boyer–Moore (BM). Learn how they work, when they perform well, and measure their efficiency on long texts.

Setup

Substring search asks: *Does pattern `P` occur in text `T`?*

- Naive algorithm: check every position → $O(n \cdot m)$.
- KMP: preprocess pattern to skip comparisons → $O(n + m)$.
- Boyer–Moore: skip ahead using heuristics → sublinear in practice.

We'll implement these in Python for clarity.

Step-by-Step

1. Naive Algorithm
    - For each position in text, compare pattern character by character.
    - Slow on long inputs with repeated matches.
2. KMP Algorithm
    - Build a failure function (longest prefix-suffix table).
    - Skip ahead when mismatch occurs.
    - Guarantees linear time.
3. Boyer–Moore Algorithm (Bad Character Rule)
    - Start matching from the end of the pattern.
    - On mismatch, shift based on last occurrence of character.
    - Very fast in practice, especially for large alphabets.
4. Benchmark on Large Input
    - Generate a text of size ~100,000 characters.
    - Search for a small pattern.
    - Measure runtimes.
5. Discussion

- Naive is simple but inefficient.
- KMP is optimal for worst case.
- Boyer–Moore is fastest in practice for natural language.

Example (Python)

```python
import time

# 1. Naive
def naive_search(text, pattern):
    n, m = len(text), len(pattern)
    for i in range(n - m + 1):
        if text[i:i+m] == pattern:
            return i
    return -1

# 2. KMP
def kmp_table(pattern):
    m = len(pattern)
    table = [0] * m
    j = 0
    for i in range(1, m):
        while j > 0 and pattern[i] != pattern[j]:
            j = table[j - 1]
        if pattern[i] == pattern[j]:
            j += 1
            table[i] = j
    return table

def kmp_search(text, pattern):
    n, m = len(text), len(pattern)
    table = kmp_table(pattern)
    j = 0
    for i in range(n):
        while j > 0 and text[i] != pattern[j]:
            j = table[j - 1]
        if text[i] == pattern[j]:
            j += 1
            if j == m:
                return i - m + 1
    return -1
```

```python
# 3. Boyer-Moore (bad character heuristic)
def bm_table(pattern):
    table = {}
    for i, c in enumerate(pattern):
        table[c] = i
    return table

def bm_search(text, pattern):
    n, m = len(text), len(pattern)
    table = bm_table(pattern)
    i = 0
    while i <= n - m:
        j = m - 1
        while j >= 0 and text[i + j] == pattern[j]:
            j -= 1
        if j < 0:
            return i
        shift = max(1, j - table.get(text[i + j], -1))
        i += shift
    return -1

# Benchmark
text = "a" * 100000 + "b"
pattern = "a" * 10 + "b"

for name, func in [("Naive", naive_search), ("KMP", kmp_search), ("Boyer-Moore", bm_search)]
    start = time.time()
    pos = func(text, pattern)
    print(f"{name} found at {pos}, time {round(time.time()-start,4)}s")
```

Expected Results

- Naive: Slow for large repetitive text.
- KMP: Linear time, handles worst case well.
- Boyer–Moore: Fastest for natural text (skips ahead aggressively).

Algorithm	Worst Case Time	Best Case Time	Notes
Naive	$O(n \cdot m)$	$O(n)$	Simple, bad for long patterns
KMP	$O(n + m)$	$O(n + m)$	Guaranteed linear time

Algorithm	Worst Case Time	Best Case Time	Notes
Boyer–Moore	$O(n \cdot m)$ worst	Sublinear avg	Very fast in practice on real text

Why it matters

- Search engines, text editors, and databases rely on efficient substring search.
- Naive methods break down at scale.
- KMP shows theory guiding better algorithms.
- Boyer–Moore demonstrates practical engineering that beats naive even further.

Exercises

1. Implement naive substring search in any language.
2. Search for `"abc"` in `"aaaabc"` and explain why naive repeats work.
3. Build the prefix table for `"ababaca"` in KMP.
4. Explain how KMP avoids re-checking characters.
5. Implement Boyer–Moore with the bad character rule.
6. Show why Boyer–Moore jumps ahead more than one character on mismatches.
7. Benchmark naive vs KMP vs BM on 1MB text.
8. Compare results for random text vs repetitive text.
9. Explain why Boyer–Moore worst-case is still $O(n \cdot m)$.
10. Propose a hybrid search strategy for DNA sequences (small alphabet, long text).

LAB 11: Regex Engine Catastrophe (ReDoS)

Goal

See how certain regex patterns cause catastrophic backtracking and turn tiny inputs into huge runtimes (Regex Denial of Service—ReDoS). Learn safe alternatives and mitigation strategies.

Setup

Any language with a backtracking engine will reproduce the issue (Python/PCRE/Perl/JavaScript). An automata-based engine (e.g., RE2, Rust's default) won't exhibit catastrophic behavior for the same patterns.

Test patterns that are notorious:

- Nested, overlapping quantifiers: `(a+)+b`
- Ambiguous alternation with stars: `(a|aa)+b`
- Catastrophic HTMLish: `(<.+>)+`

Step-by-Step

1. Reproduce a Fast Match
    - Pattern: `(a+)+b`
    - Input: `"a"*25 + "b"`
    - Expect: quick success (engine finds the trailing `b`).
2. Trigger Catastrophe
    - Same pattern: `(a+)+b`
    - Input: `"a"*25` (no `b`)
    - Expect: very slow—engine explores exponential backtracking paths.
3. Measure Impact
    - Time both inputs.
    - Increase input length stepwise (e.g., 15, 20, 25, 30 `a`s).
    - Observe runtime explosion for the no-`b` case.
4. Try Another Problematic Pattern
    - Pattern: `(a|aa)+b`
    - Inputs: `"a"*N+"b"` (fast) and `"a"*N` (slow).
    - Note that overlapping alternatives also trigger exponential blowups.
5. Mitigate
    - Refactor to unambiguous patterns:
        - Replace `(a+)+b` with `a+ b`-style constructions that avoid nested quantifiers, or use possessive/atomic quantifiers if supported (e.g., `a++b`, `(?>a+)b`).
    - Constrain with anchors and character classes to reduce search space.
    - Precompile and timeout: set execution time limits (if supported).
    - Use safe engines (RE2/Hyperscan) for untrusted input.

Example (Python)

```python
import re, time

def timed_match(pattern, text, flags=0):
    start = time.time()
    ok = bool(re.match(pattern, text, flags))
    dur = time.time() - start
    return ok, dur

# 1) Fast match: pattern matches because of trailing 'b'
pat = r"(a+)+b"
fast_text = "a" * 25 + "b"
ok, dur = timed_match(pat, fast_text)
print("FAST   -> match:", ok, "time:", round(dur, 4), "s")

# 2) Catastrophic: same pattern, missing the final 'b'
slow_text = "a" * 25
ok, dur = timed_match(pat, slow_text)
print("SLOW   -> match:", ok, "time:", round(dur, 4), "s")

# 3) Scale up to see blowup
for n in [20, 22, 24, 26, 28]:
    txt = "a" * n
    ok, dur = timed_match(pat, txt)
    print(f"N={n:<2} -> match:{ok} time:{round(dur, 4)} s")

# 4) Another problematic pattern
pat2 = r"(a|aa)+b"
ok, dur = timed_match(pat2, "a" * 25 + "b")
print("ALT OK -> match:", ok, "time:", round(dur, 4), "s")

ok, dur = timed_match(pat2, "a" * 25)
print("ALT SLOW-> match:", ok, "time:", round(dur, 4), "s")

# 5) Safer alternative using atomic/possessive (if your engine supports it)
# Python's 're' does not support atomic groups/possessive quantifiers.
# Pseudocode examples for other engines:
#   - PCRE/Java: r"(?>a+)b" or r"a++b"
# For Python, restructure logic instead of relying on engine features.
```

Expected Results

- With trailing `b`, matches are fast.
- Without `b`, runtime grows rapidly as `N` increases.
- Overlapping alternations show the same pattern of slowdown.
- Atomic/possessive quantifiers (in engines that support them) remove backtracking and restore predictable performance.

Pattern	Input	Engine Type	Behavior	
`(a+)+b`	`a…ab`	Backtracking	Fast	
`(a+)+b`	`a…a`	Backtracking	Catastrophic (very slow)	
‘(a	aa)+b‘	`a…ab`	Backtracking	Fast
‘(a	aa)+b‘	`a…a`	Backtracking	Catastrophic
`(?>a+)b`	`a…a?`	Atomic (PCRE/Java)	Predictable, no catastrophe	
`a++b`	`a…a?`	Possessive	Predictable	
RE2 version	any of above	DFA/NFA (safe)	Linear-time, no catastrophe	

Why it matters

- Untrusted input + fragile regex = DoS vector.
- Web apps, API gateways, and log pipelines often apply regex to attacker-controlled text.
- The fix is architectural (pick safe engines) and design-oriented (write non-ambiguous patterns, set timeouts).

Exercises

1. Explain why `(a+)+b` is dangerous on inputs composed only of `a`s.
2. Show a timing table for `(a+)+b` against `a^N` for N = 10, 15, 20, 25.
3. Construct another catastrophic pattern using overlapping alternation (e.g., `(ab|a)*b`).
4. Rewrite `(a|aa)+b` into a pattern that avoids ambiguity while matching the same language.
5. Describe how atomic groups or possessive quantifiers prevent backtracking.
6. Propose engine-agnostic mitigations: anchoring, limiting input length, pre-validation.
7. Design a test harness that detects suspicious regex (runtime or backtracking depth spikes).
8. Compare behavior of the same pattern in a backtracking engine vs a linear-time engine (e.g., RE2).
9. For a production route-matching regex, list safeguards to prevent ReDoS in a web server.
10. Outline a policy for your organization: when to allow backreferences/lookbehinds; when to mandate RE2-class engines.

LAB 12: Greedy vs Lazy Matching

Goal

See how regex quantifiers (`*`, `+`, `?`) behave in greedy mode (match as much as possible) versus lazy mode (match as little as possible). Learn when greedy matching leads to over-capture and how lazy mode fixes it.

Setup

Regex engines default to greedy quantifiers:

- `.*` matches the longest possible string.
- Adding `?` makes them lazy: `.*?` matches the shortest possible string.

We'll use Python's `re` module.

Step-by-Step

1. Simple Greedy Match
    - Pattern: `<tag>.*</tag>`
    - Input: `"<tag>one</tag><tag>two</tag>"`
    - Greedy captures from the first `<tag>` to the last `</tag>`, swallowing too much.
2. Switch to Lazy
    - Pattern: `<tag>.*?</tag>`
    - Same input.
    - Lazy captures each `<tag>…</tag>` pair separately, as expected.
3. Experiment with Plus Quantifier
    - Compare `<tag>.+</tag>` vs `<tag>.+?</tag>`.
    - `+` requires at least one character, `*` allows zero.
4. Realistic Example: HTML-ish Parsing
    - Input: `"<b>bold</b><i>italic</i>"`.
    - Greedy regex: `<.*>` → captures everything between first `<` and last `>`.
    - Lazy regex: `<.*?>` → captures `<b>`, `</b>`, `<i>`, `</i>` separately.
5. Discussion

- Greedy matching is fine when only one block exists.
- Lazy matching is safer for repeated structures.
- Regex is not a full parser; overuse leads to fragile code.

Example (Python)

```
import re

text = "<tag>one</tag><tag>two</tag>"

# 1. Greedy match
greedy = re.findall(r"<tag>.*</tag>", text)
print("Greedy:", greedy)

# 2. Lazy match
lazy = re.findall(r"<tag>.*?</tag>", text)
print("Lazy:", lazy)

# 3. Greedy vs lazy with plus quantifier
sample = "<tag>a</tag><tag>b</tag>"
print("Greedy +:", re.findall(r"<tag>.+</tag>", sample))
print("Lazy +?:", re.findall(r"<tag>.+?</tag>", sample))

# 4. HTML-ish example
html = "<b>bold</b><i>italic</i>"
print("Greedy HTML:", re.findall(r"<.*>", html))
print("Lazy HTML:", re.findall(r"<.*?>", html))
```

Expected Results

- Greedy consumes everything between first and last match.
- Lazy matches each block separately.

Pattern	Input	Output
`<tag>.*</tag>`	`<tag>one</tag><tag>two</tag>`	`['<tag>one</tag><tag>two</tag>']`
`<tag>.*?</tag>`	`<tag>one</tag><tag>two</tag>`	`['<tag>one</tag>', '<tag>two</tag>']`
`<.*>` (greedy)	`<b>bold</b><i>italic</i>`	`['<b>bold</b><i>italic</i>']`
`<.*?>` (lazy)	`<b>bold</b><i>italic</i>`	`['<b>', '</b>', '<i>', '</i>']`

Why it matters

- Greedy vs lazy matching changes how much text is captured.
- Incorrect choice can swallow entire documents or miss intended matches.
- Useful in log parsing, HTML scraping, and template matching.
- Shows why regex is powerful but also error-prone when used without care.

Exercises

1. Write a regex to capture `<tag>…</tag>` in `"<tag>a</tag><tag>b</tag>"`. Compare greedy vs lazy.
2. Show why `<.*>` over-captures in `"<p>hi</p><p>bye</p>"`.
3. Modify `<.*>` into `<.*?>` and explain the difference.
4. Use `.+?` to capture non-empty blocks. Show how it differs from `.*?`.
5. Write a regex that extracts `<b>…</b>` text only.
6. Compare results of `<div>.*</div>` on a file with nested `<div>` tags.
7. Explain why lazy quantifiers may still produce unexpected results in deeply nested structures.
8. Benchmark `.*` vs `.*?` on a 1MB HTML file.
9. Show how greedy vs lazy affects performance (number of backtracking steps).
10. Explain why full HTML parsing should not rely solely on regex.

LAB 13: Build a Mini Inverted Index

Goal

Implement a simplified inverted index, the core data structure behind search engines. Learn how to map words to the documents they appear in, then use the index to answer queries quickly.

Setup

An inverted index stores entries like:

```
"dog" → [doc1, doc3]
"cat" → [doc2]
```

Instead of scanning every document, you look up the word in the index. We'll implement this in Python, but the idea applies to any language or system.

Step-by-Step

1. Prepare Documents
    - Create a small collection of texts, each with an ID.
    - Example:
        - `1: "the quick brown fox"`
        - `2: "the lazy dog"`
        - `3: "the fox jumped over the dog"`
2. Tokenize Documents
    - Split text into lowercase words.
    - Remove punctuation (simplified tokenizer).
3. Build Inverted Index
    - For each word, append the document ID to its list.
    - Use a dictionary (map) of word → set of doc IDs.
4. Query the Index
    - Look up a word and return all documents containing it.
    - Extend to multi-word queries by intersecting sets.
5. Discussion
    - Inverted index allows fast search compared to scanning.
    - Real search engines add ranking (TF-IDF, BM25), phrase search, and indexing optimizations.

Example (Python)

```
from collections import defaultdict
import re

# 1. Documents
docs = {
    1: "the quick brown fox",
    2: "the lazy dog",
    3: "the fox jumped over the dog"
}
```

```
# 2. Tokenize
def tokenize(text):
    return re.findall(r"\w+", text.lower())

# 3. Build inverted index
index = defaultdict(set)
for doc_id, text in docs.items():
    for word in tokenize(text):
        index[word].add(doc_id)

# 4. Query
def search(word):
    return index.get(word.lower(), set())

def search_multi(words):
    sets = [search(w) for w in words]
    return set.intersection(*sets) if sets else set()

# Example queries
print("Index for 'fox':", index["fox"])
print("Search 'dog':", search("dog"))
print("Search 'fox AND dog':", search_multi(["fox", "dog"]))
```

Expected Results

- Index for `"fox"` → `{1, 3}`
- Search `"dog"` → `{2, 3}`
- Search `"fox AND dog"` → `{3}`

Word	Document IDs
the	{1, 2, 3}
quick	{1}
brown	{1}
fox	{1, 3}
lazy	{2}
dog	{2, 3}
jumped	{3}
over	{3}

Why it matters

- This is the foundation of Google, Lucene, Elasticsearch and every major search system.
- Inverted indexes make keyword search efficient.
- They scale from a few documents to billions.
- Concepts here connect to IR (Information Retrieval) theory.

Exercises

1. Build an inverted index for 5 custom sentences.
2. Search for a single word and list document IDs.
3. Implement AND search: return docs containing all words.
4. Implement OR search: return docs containing at least one word.
5. Extend the tokenizer to remove stop words like `"the"`, `"and"`.
6. Modify the index to count word frequency in each document.
7. Add a simple ranking: prefer documents with more matches.
8. Test the system with queries on `"cat"` when no doc contains `"cat"`.
9. Explain why inverted indexes are better than scanning.
10. Design how you would scale this to 1 million documents.

LAB 14: Compression of Repetitive Strings

Goal

Understand how string compression reduces storage by exploiting repetition. Learn simple dictionary-based compression, compare sizes with and without compression, and see how algorithms like gzip or Huffman coding achieve savings.

Setup

We'll simulate a simple compression scheme:

- Naive storage: keep the string as-is.
- Dictionary-based compression: replace repeated substrings with references.
- Compare results with Python's `zlib` (which implements DEFLATE, combining LZ77 + Huffman coding).

Test input:

```
"banana banana banana"
```

Step-by-Step

1. Uncompressed Storage
    - Measure length of original string in bytes.
2. Naive Dictionary Compression
    - Store `"banana"` once.
    - Represent repeated `"banana"`s as references.
    - Calculate compressed size (dictionary + references).
3. zlib Compression
    - Use `zlib.compress()` to compress.
    - Compare compressed size with original.
4. Test with Different Inputs
    - Highly repetitive: `"abc abc abc abc"` → compresses well.
    - Random: `"qwertyuiop"` → compresses poorly.
5. Discussion
    - Compression works best when redundancy exists.
    - Dictionary + entropy coding are the backbone of text and file compression.
    - Trade-off: compression saves space but costs CPU time.

Example (Python)

```
import zlib

# 1. Original string
text = "banana banana banana"
original_size = len(text.encode("utf-8"))
print("Original text:", text)
print("Original size:", original_size, "bytes")

# 2. Naive dictionary compression simulation
dictionary = {"banana": 1}
tokens = [dictionary.get(word, word) for word in text.split()]
compressed_repr = (dictionary, tokens)
simulated_size = len("banana") + len(tokens)  # rough size
```

```
print("Naive simulated compressed size:", simulated_size, "bytes")

# 3. zlib compression
compressed = zlib.compress(text.encode("utf-8"))
print("zlib compressed size:", len(compressed), "bytes")

# 4. Compare with random string
random_text = "qwertyuiop"
compressed_random = zlib.compress(random_text.encode("utf-8"))
print("Random original size:", len(random_text), "bytes")
print("Random compressed size:", len(compressed_random), "bytes")
```

Expected Results

- Original: ~20 bytes.
- Naive dictionary scheme: fewer bytes than original.
- zlib: compressed size significantly smaller (e.g., 20 → 17).
- Random text: compressed size original (sometimes slightly larger due to overhead).

Input	Original Size	Compressed Size
`"banana banana banana"`	20 bytes	~17 bytes
`"abc abc abc abc"`	15 bytes	~13 bytes
`"qwertyuiop"`	10 bytes	~12 bytes (overhead)

Why it matters

- Compression underpins storage (zip files, databases, backups) and transmission (HTTP gzip, messaging).
- Saves bandwidth and storage by removing redundancy.
- Shows trade-offs: repetitive text compresses well, random text does not.
- Foundation for deeper algorithms: Huffman coding, LZ77, arithmetic coding.

Exercises

1. Compute original and compressed sizes of `"hello hello hello"`.
2. Compare compression ratio for `"aaaaa"` vs `"abcde"`.
3. Write a function that stores unique words in a dictionary and replaces repeats with indexes.

4. Extend your dictionary to store `"banana"` as key `1` and `"apple"` as key `2`. Compress `"banana apple banana"`.
5. Measure zlib compression ratio for a large string of repeated `"test"`.
6. Generate random strings of length 100 and test compression. What happens?
7. Explain why compression sometimes increases size on short random inputs.
8. Compare zlib compression with `bz2` or `lzma` in Python.
9. Explain how compression is used in network protocols like HTTP/2.
10. Propose how compression could be applied in search engine indexes (from LAB 13).

LAB 15: Injection Attack Simulation

Goal

Demonstrate how insecure string handling in queries can lead to injection attacks (e.g., SQL injection). Show how malicious inputs manipulate string-based queries, and how parameterized queries prevent this.

Setup

Any system that builds queries with string concatenation is vulnerable. We'll simulate SQL injection in Python:

- Unsafe example: concatenating user input directly into a query string.
- Safe example: using parameterized queries (`?` or `%s`).

We'll use `sqlite3` for demonstration, since it's lightweight and built into Python.

Step-by-Step

1. Create a Sample Database
   - Table: `accounts(name TEXT, balance INT)`.
   - Insert rows: `"Alice", 100`, `"Bob", 50`.
2. Unsafe Query
   - Take user input for name.
   - Build query with string concatenation:
     ```
     SELECT * FROM accounts WHERE name = 'USER_INPUT';
     ```
   - Input `"Alice"` → works fine.

- Input `"' OR '1'='1"` → query returns all rows (attack).

3. Simulated SQL Injection Attack
   - Input `"; DROP TABLE accounts; --"` → destructive query.
   - In real systems, this would delete the table.
4. Safe Query with Parameters
   - Use `cursor.execute("SELECT * FROM accounts WHERE name = ?", (user_input,))`.
   - Input `"' OR '1'='1"` → safely treated as a literal string, no injection.
5. Discussion
   - String concatenation = dangerous.
   - Parameterization = safe.
   - This applies to SQL, shell commands, HTML, and beyond.

Example (Python)

```
import sqlite3

# 1. Setup
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute("CREATE TABLE accounts (name TEXT, balance INT)")
c.executemany("INSERT INTO accounts VALUES (?, ?)", [("Alice", 100), ("Bob", 50)])
conn.commit()

# 2. Unsafe query
def unsafe_query(user_input):
    query = f"SELECT * FROM accounts WHERE name = '{user_input}'"
    print("Unsafe query:", query)
    return list(c.execute(query))

# Safe query
def safe_query(user_input):
    query = "SELECT * FROM accounts WHERE name = ?"
    print("Safe query:", query)
    return list(c.execute(query, (user_input,)))

# Normal input
```

```
print("Normal unsafe:", unsafe_query("Alice"))

# Injection attack
print("Injection unsafe:", unsafe_query("' OR '1'='1"))

# Safe version prevents injection
print("Safe version:", safe_query("' OR '1'='1"))
```

Expected Results

- Normal input `"Alice"` works fine.
- Injection input `"' OR '1'='1"` returns all rows in unsafe query.
- Safe query treats it literally → no injection.

Input	Unsafe Query Result	Safe Query Result
`"Alice"`	`[("Alice", 100)]`	`[("Alice", 100)]`
`"' OR '1'='1"`	`[("Alice",100),("Bob",50)]`	`[]` (no match)

Why it matters

- SQL injection is one of the most critical web vulnerabilities (OWASP Top 10).
- Attackers can dump data, bypass authentication, or destroy databases.
- The problem is not SQL itself but string misuse.
- Fix: always use parameterized queries or ORMs with safe bindings.

Exercises

1. Write a query vulnerable to injection and test with `"Alice"`.
2. Inject `"' OR '1'='1"` and explain why it returns all rows.
3. Try input `"; DROP TABLE accounts; --"` and explain what would happen in a real DB.
4. Rewrite your code to use parameterized queries.
5. Test injection again with safe queries and verify no effect.
6. Extend this idea to shell commands (`os.system`) and simulate command injection.
7. Research one real-world breach caused by SQL injection.
8. Explain why escaping input manually (e.g., replacing quotes) is insufficient.
9. Design input validation rules that complement parameterization.
10. Propose a security guideline for handling strings in queries for your organization.