https://simplifycpp.org

DRAFT

Essential Algorithms in Rust

Practical Engineering Handbook

Prepared by Ayman Alheraki

Essential Algorithms in Rust

Practical Engineering Handbook

Some drafting assistance and idea exploration were supported by modern AI tools, with full supervision, verification, correction, and authorship.

Prepared by Ayman Alheraki

February 2026

Contents

Author's Introduction

Algorithms are the true foundation of software engineering. They determine the quality of a solution, its efficiency, its scalability, and its long-term reliability. Without a deep understanding of algorithms, programming becomes merely writing instructions; with it, programming becomes disciplined engineering.

I chose **Rust** for this book because it enforces strict memory management and structural discipline, making the implementation of algorithms a powerful learning experience that combines high performance with strong safety guarantees. This discipline encourages correct thinking from the very beginning, aligning perfectly with the philosophy of this work.

This book is designed to serve as a concise and practical reference focused on essential algorithmic patterns, supported by clear examples and precise complexity and performance analysis. The goal is not memorization, but understanding and sound engineering application.

I hope this handbook becomes a reliable companion that strengthens your algorithmic foundation and equips you with practical tools for building robust solutions in Rust.

Ayman Alheraki

Preface

Why This Book Exists

This booklet exists to close a practical gap:

- Many algorithm books teach ideas, but not the engineering decisions required to ship correct, maintainable, and measurable implementations.
- Many Rust books teach the language, but not how to systematically translate an algorithmic idea into a tested, benchmarked, production-worthy component.

Algorithms are not just "a trick that passes an interview." In real engineering work, algorithms are the core of:

- performance and latency budgets,
- memory behavior and predictability,
- correctness under edge cases and adversarial inputs,
- maintainability and refactoring safety.

This handbook therefore treats algorithms as *components*:

- a clear API,

- explicit invariants,
- tests that prove the invariants hold,
- benchmarks that quantify tradeoffs,
- documentation that prevents misuse.

A concrete example of what “engineering algorithm” means

You can learn binary search in one paragraph. But a reliable binary search utility is a small engineering artifact:

- it must define what happens when the key is absent,
- it must handle duplicates deterministically (first/last occurrence),
- it must prove termination and correctness,
- it must avoid overflow in index arithmetic,
- it must be tested with edge cases and fuzz-like randomized checks.

```rust
/// Returns the leftmost index i such that a[i] >= key (a must be sorted).
/// This is "lower_bound".
pub fn lower_bound<T: Ord>(a: &[T], key: &T) -> usize {
    let mut lo: usize = 0;
    let mut hi: usize = a.len(); // invariant: answer is in [lo, hi]
    while lo < hi {
        // mid is safe from overflow: lo + (hi - lo)/2
        let mid = lo + (hi - lo) / 2;
        if &a[mid] < key {
            lo = mid + 1;
        } else {
```

```
            hi = mid;
        }
    }
    lo
}

#[cfg(test)]
mod tests {
    use super::lower_bound;

    #[test]
    fn lower_bound_basic() {
        let a = [1, 2, 2, 2, 5, 9];
        assert_eq!(lower_bound(&a, &0), 0);
        assert_eq!(lower_bound(&a, &1), 0);
        assert_eq!(lower_bound(&a, &2), 1);
        assert_eq!(lower_bound(&a, &3), 4);
        assert_eq!(lower_bound(&a, &10), a.len());
    }
}
```

This is the style of this book: simple idea → explicit contract → implementation → tests.

Why Rust Changes the Conversation

Rust changes algorithm engineering because it forces you to make correctness and ownership explicit at compile time.

1) Correctness becomes structural

Rust encourages building APIs that prevent misuse:

- ownership clarifies who may mutate,
- borrowing clarifies lifetimes of references,
- pattern matching forces case coverage,
- enums replace sentinel values,
- results replace implicit error paths.

2) Performance stays first-class

Rust aims for predictable performance:

- no mandatory GC pauses,
- efficient value types,
- control over allocation,
- iterators can compile to tight loops,
- explicit bounds checks that the optimizer can often eliminate.

3) “Unsafe” is isolated

Some algorithms require low-level tricks (SIMD, manual memory layouts, custom allocators). Rust allows that, but encourages *confinement*:

- keep `unsafe` inside a tiny module,
- expose a safe API with documented invariants,
- test heavily at the boundaries.

A small example: choosing safety by default

Many algorithm implementations in other languages silently accept invalid indices or invalid states. Rust nudges you toward explicit state models:

```rust
/// A safe "stack" with explicit operations and no invalid states.
pub struct Stack<T> {
    v: Vec<T>,
}

impl<T> Stack<T> {
    pub fn new() -> Self { Self { v: Vec::new() } }
    pub fn push(&mut self, x: T) { self.v.push(x); }
    pub fn pop(&mut self) -> Option<T> { self.v.pop() }
    pub fn peek(&self) -> Option<&T> { self.v.last() }
    pub fn len(&self) -> usize { self.v.len() }
    pub fn is_empty(&self) -> bool { self.v.is_empty() }
}
```

That may look trivial, but the same discipline scales to graphs, DP tables, heaps, and indexing-heavy code.

Scope and Philosophy

This booklet is a **practical engineering handbook**.

Scope

We focus on algorithms that are repeatedly used in real systems:

- searching and selection (binary search, lower/upper bounds, quickselect),
- sorting and ordering (including stability, custom comparators),

- hashing fundamentals and collision realities,
- stacks/queues/deques and monotonic structures,
- heaps and priority queues,
- union-find (disjoint sets),
- graphs (BFS/DFS, shortest paths, MST basics),
- dynamic programming patterns (tabulation, memoization, state compression),
- string algorithms (prefix function / KMP ideas, rolling hash patterns),
- numeric and bit tricks that improve constant factors safely.

Each topic is treated as:

- a contract: input assumptions, output guarantees,
- an invariant: what must remain true at every step,
- a complexity statement: time and memory,
- an implementation: idiomatic Rust,
- a test suite: edge cases and randomized checks when appropriate,
- a benchmark note: what to measure and why.

Philosophy

- **Clarity first, then speed.** A fast bug is still a bug.
- **Make invariants visible.** Use names, types, and comments that encode the proof idea.

- **Measure, don't guess.** Performance is a data problem, not a belief.
- **Prefer safe Rust.** Use `unsafe` only when a measurable win exists and invariants are documented.
- **Engineer the interface.** A correct algorithm behind a confusing API is still a liability.

What This Book Is Not

To keep this booklet sharp, here is what it intentionally does *not* try to do:

- It is not a full Rust language course.
- It is not an academic textbook that proves every theorem formally.
- It is not a competitive programming cheat sheet.
- It is not a catalog of every known algorithm.
- It is not a crate-by-crate tour of the Rust ecosystem.

You will still learn Rust by reading and practicing the code here, but the goal is algorithmic engineering, not language coverage.

How to Use This Book

Use this book like a workshop manual.

Recommended workflow

1. Read the contract and invariants before reading the code.
2. Re-implement the algorithm from scratch without copying.
3. Add at least 5 edge-case tests of your own.
4. Run benchmarks after you have a correct baseline.
5. Only then consider micro-optimizations.

Windows-first Rust setup (cargo workflow)

All examples assume a normal Rust toolchain on Windows and are designed to run with `cargo`.

```
# Create a new project
cargo new algo_lab
cd algo_lab

# Run tests
cargo test

# Run in release mode (important for benchmarks and performance)
cargo run --release

# Check formatting and linting discipline
cargo fmt
cargo clippy -- -D warnings
```

How to read the code

In this book:

- functions are small and single-purpose,
- preconditions are stated clearly,
- invariants are mentioned near the loop,
- tests come with both fixed cases and (sometimes) randomized validation.

A suggested folder structure

As your implementations grow, split them into modules:

```
src/
  lib.rs
  search.rs
  sort.rs
  heap.rs
  graph.rs
  dp.rs
tests/
  randomized.rs
benches/
  perf.rs
```

Intended Audience

This booklet is for:

- software engineers who already code professionally and want stronger algorithmic foundations,
- systems programmers who want predictable performance with safer interfaces,

- Rust developers who want a structured approach to algorithms beyond “I saw it once,”
- students who want engineering-quality implementations, not only theory.

Prerequisites

You should be comfortable with:

- basic Rust syntax (`fn`, `struct`, `enum`, borrowing),
- basic complexity intuition ($O(n)$ vs $O(n \log n)$),
- reading tests and understanding failure output.

If you are completely new to Rust, you can still use this book, but expect to pause and learn language fundamentals in parallel.

Engineering Discipline Over Memorization

Memorization fails in real engineering because the problem shape changes:

- different constraints,
- different input distributions,
- different correctness requirements,
- different memory budgets,
- different concurrency or latency conditions.

Discipline scales. This book trains discipline through a repeatable checklist.

The algorithm engineering checklist

For each algorithm you implement, you should be able to answer:

1. **Contract:** What do I accept, and what do I guarantee?
2. **Invariants:** What must stay true after every iteration?
3. **Edge cases:** Empty input, one element, duplicates, max sizes, adversarial shapes.
4. **Complexity:** Time and memory, including constants and allocations.
5. **Failure modes:** Panics, overflow, invalid indices, recursion depth.
6. **Tests:** Deterministic unit tests plus randomized checks where useful.
7. **Measurement:** What metric matters (throughput, latency, memory, allocations)?

Example: discipline with randomized validation

Below is a tiny pattern you will see repeatedly: verify a specialized algorithm against a simple baseline.

```rust
pub fn is_sorted<T: Ord>(a: &[T]) -> bool {
    a.windows(2).all(|w| w[0] <= w[1])
}

#[cfg(test)]
mod randomized {
    use super::is_sorted;

    // A tiny deterministic pseudo-random generator (no external crates).
    fn next_u64(state: &mut u64) -> u64 {
        // LCG constants (good enough for tests, not for crypto)
```

```
        *state = state.wrapping_mul(6364136223846793005).wrapping_add(1);
        *state
    }

    #[test]
    fn is_sorted_matches_baseline_on_random_data() {
        let mut seed = 0x1234_5678_9abc_def0u64;

        for _ in 0..200 {
            let n = (next_u64(&mut seed) % 64) as usize;
            let mut v: Vec<i32> = (0..n)
                .map(|_| (next_u64(&mut seed) % 200) as i32 - 100)
                .collect();

            // Baseline: sort then check.
            v.sort();
            assert!(is_sorted(&v));

            // Perturb: swap if possible, might break sortedness.
            if v.len() >= 2 {
                let i = (next_u64(&mut seed) % v.len() as u64) as usize;
                let j = (next_u64(&mut seed) % v.len() as u64) as usize;
                v.swap(i, j);

                // Baseline check: compare to the simplest definition.
                let baseline = v.windows(2).all(|w| w[0] <= w[1]);
                assert_eq!(is_sorted(&v), baseline);
            }
        }
    }
}
```

This approach is not about being clever. It is about being *reliable*.

Final note

If you finish this book with one habit, let it be this:

> Never trust an algorithm implementation until you have (1) stated the invariants, (2) tested edge cases, and (3) measured what matters in release mode.

Rust Essentials for Algorithm Design

1.1 Project Setup with Cargo

Rust algorithm work becomes reliable when you treat it as an engineering project: clear module layout, strict tests, and repeatable builds on Windows. This section gives a Windows-first setup that supports:

- reusable implementations in `src/lib.rs`,
- small demos in `src/main.rs`,
- unit and integration tests,
- release-mode runs for realistic performance measurements.

Create a new project (Windows PowerShell)

```
cargo new algo_handbook
cd algo_handbook
```

Turn it into a library-first project

Even if you keep a `main.rs` for demos, your algorithms should live in a library so they can be tested and reused.

```
# Ensure lib.rs exists (Cargo auto-detects a library when src/lib.rs exists)
New-Item -ItemType File -Path .\src\lib.rs -Force
```

Recommended layout:

```
algo_handbook/
  Cargo.toml
  src/
    lib.rs
    main.rs          (optional)
    search.rs
    sort.rs
    heap.rs
  tests/
    integration.rs   (optional integration tests)
```

A strict but practical Cargo.toml

Release profile choices strongly impact algorithm benchmarks. Use `-release` when measuring. The options below are common for performance-oriented builds.

```
[package]
name = "algo_handbook"
version = "0.1.0"
edition = "2021"

[profile.release]
opt-level = 3
lto = true
codegen-units = 1
panic = "abort"
```

Core cargo commands (Windows)

```
# Build
cargo build

# Test
cargo test

# Run in release mode
cargo run --release

# Test in release mode (important for perf-sensitive code paths)
cargo test --release

# Format and lint
cargo fmt
cargo clippy -- -D warnings
```

Minimal module wiring

```
/* src/lib.rs */
pub mod search;
pub mod heap;
pub mod maps;

pub use search::*;
pub use heap::*;
pub use maps::*;
```

1.2 Ownership Rules that Impact Performance

Ownership is a performance tool. It forces you to make explicit decisions about:

- when data is moved vs borrowed,
- when heap allocation occurs,
- when deep copies (`clone`) happen,
- whether an algorithm can be in-place (`&mut [T]`) or must allocate.

Borrow slices for read-only algorithms

If an algorithm only reads input, prefer `&[T]` over `Vec<T>`. This avoids forcing the caller to allocate a vector and avoids moving ownership.

```
/* src/search.rs */
pub fn sum_i64(a: &[i64]) -> i64 {
    let mut s = 0i64;
    for &x in a {
        s += x;
    }
    s
}
```

Use `&mut [T]` for in-place algorithms

In-place algorithms reduce allocations and improve cache locality.

```
/* Reverse in place */
pub fn reverse_in_place<T>(a: &mut [T]) {
```

```
    let mut i = 0usize;
    let mut j = a.len();
    while i < j {
        j -= 1;
        a.swap(i, j);
        i += 1;
    }
}
```

Understand Copy vs Move vs Clone

Copy types (integers, small scalars) are copied implicitly. Heap-owning types (`String`, `Vec<T>`) are moved by default; cloning duplicates heap data.

```
pub fn copy_move_clone_demo() {
    let x: u64 = 7;
    let y = x;          // Copy: x still usable
    let _ = (x, y);

    let s1 = String::from("abc");
    let s2 = s1;         // Move: s1 no longer usable
    let s3 = s2.clone(); // Clone: deep copy
    let _ = (s3,);
}
```

Avoid accidental cloning in hot paths

A common algorithm slow-down is cloning large keys or records. Prefer comparing references, not allocating new objects.

```
/* binary search membership without cloning */
```

```rust
pub fn contains_sorted<T: Ord>(a: &[T], key: &T) -> bool {
    let mut lo = 0usize;
    let mut hi = a.len();
    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if &a[mid] < key {
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    lo < a.len() && &a[lo] == key
}
```

1.3 Efficient Use of `Vec` and Slices

`Vec<T>` is the primary container for algorithm building blocks. Efficiency often comes from:

- capacity planning,
- minimizing reallocations,
- avoiding repeated bounds checks,
- using slices to avoid copies and support subranges.

Pre-allocate capacity when possible

If you can estimate output size, `with_capacity` reduces reallocations.

```rust
pub fn filter_non_negative(a: &[i32]) -> Vec<i32> {
    let mut out = Vec::with_capacity(a.len());
```

```
    for &x in a {
        if x >= 0 {
            out.push(x);
        }
    }
    out
}
```

Prefer slice APIs to maximize reuse

Slice-based functions accept vectors, arrays, and sub-slices without copying.

```
pub fn rotate_left_by_one<T>(a: &mut [T]) {
    if a.len() <= 1 { return; }
    a.rotate_left(1);
}
```

Use slice combinators for clarity and predictable bounds checks

Methods like `windows` and `chunks` express intent and often avoid manual indexing complexity.

```
pub fn is_sorted_i32(a: &[i32]) -> bool {
    a.windows(2).all(|w| w[0] <= w[1])
}
```

Two-pointer patterns on slices

Two-pointer designs are common in algorithm engineering and map naturally to slices.

```rust
/* Merge step: merge two sorted slices into a new Vec */
pub fn merge_sorted(a: &[i32], b: &[i32]) -> Vec<i32> {
    let mut out = Vec::with_capacity(a.len() + b.len());
    let (mut i, mut j) = (0usize, 0usize);
    while i < a.len() && j < b.len() {
        if a[i] <= b[j] {
            out.push(a[i]); i += 1;
        } else {
            out.push(b[j]); j += 1;
        }
    }
    out.extend_from_slice(&a[i..]);
    out.extend_from_slice(&b[j..]);
    out
}
```

1.4 Quick Introduction to `HashMap` and `BinaryHeap`

Rust standard collections cover most day-to-day algorithm needs.

`HashMap`: counting and indexing

```rust
/* src/maps.rs */
use std::collections::HashMap;

pub fn frequency(words: &[&str]) -> HashMap<String, usize> {
    let mut m: HashMap<String, usize> = HashMap::new();
    for &w in words {
        *m.entry(w.to_string()).or_insert(0) += 1;
    }
    m
}
```

`HashMap`: memoization pattern

```rust
use std::collections::HashMap;

pub fn fib(n: u64) -> u64 {
    fn go(n: u64, memo: &mut HashMap<u64, u64>) -> u64 {
        if let Some(&v) = memo.get(&n) { return v; }
        let v = match n {
            0 => 0,
            1 => 1,
            _ => go(n - 1, memo) + go(n - 2, memo),
        };
        memo.insert(n, v);
        v
    }
    let mut memo = HashMap::new();
    go(n, &mut memo)
}
```

`BinaryHeap`: top-k elements (max-heap)

```rust
/* src/heap.rs */
use std::collections::BinaryHeap;

pub fn top_k(a: &[i32], k: usize) -> Vec<i32> {
    let mut heap = BinaryHeap::new();
    for &x in a {
        heap.push(x);
    }
    let k = k.min(heap.len());
    (0..k).filter_map(|_| heap.pop()).collect()
}
```

Min-heap behavior using `Reverse`

```rust
use std::cmp::Reverse;
use std::collections::BinaryHeap;

pub fn k_smallest(a: &[i32], k: usize) -> Vec<i32> {
    let mut heap: BinaryHeap<Reverse<i32>> = BinaryHeap::new();
    for &x in a {
        heap.push(Reverse(x));
    }
    let k = k.min(heap.len());
    let mut out: Vec<i32> = (0..k)
        .filter_map(|_| heap.pop())
        .map(|r| r.0)
        .collect();
    out.sort();
    out
}
```

1.5 Writing Clean Unit Tests

Algorithm implementations should be tested as a first-class artifact. Clean tests have:

- clear intent,
- small input cases that isolate behavior,
- explicit edge cases,
- optional randomized validation against a baseline reference.

Deterministic unit tests

```rust
pub fn lower_bound_i32(a: &[i32], key: i32) -> usize {
    let mut lo = 0usize;
    let mut hi = a.len();
    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if a[mid] < key {
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    lo
}

#[cfg(test)]
mod tests {
    use super::lower_bound_i32;

    #[test]
    fn empty() {
        let a: [i32; 0] = [];
        assert_eq!(lower_bound_i32(&a, 7), 0);
    }

    #[test]
    fn duplicates() {
        let a = [1, 2, 2, 2, 5];
        assert_eq!(lower_bound_i32(&a, 2), 1);
        assert_eq!(lower_bound_i32(&a, 3), 4);
    }
```

```
    #[test]
    fn extremes() {
        let a = [10, 20, 30];
        assert_eq!(lower_bound_i32(&a, -1), 0);
        assert_eq!(lower_bound_i32(&a, 999), 3);
    }
}
```

Randomized validation against a baseline

```
fn baseline_lower_bound(a: &[i32], key: i32) -> usize {
    for i in 0..a.len() {
        if a[i] >= key { return i; }
    }
    a.len()
}

#[cfg(test)]
mod randomized {
    use super::{baseline_lower_bound, lower_bound_i32};

    fn next_u32(s: &mut u32) -> u32 {
        s.wrapping_mul(1664525).wrapping_add(1013904223)
    }

    #[test]
    fn matches_baseline_many_times() {
        let mut seed = 1u32;
        for _ in 0..300 {
            seed = next_u32(&mut seed);
            let n = (seed % 64) as usize;
```

```
            let mut v: Vec<i32> = Vec::with_capacity(n);
            for _ in 0..n {
                seed = next_u32(&mut seed);
                v.push((seed % 200) as i32 - 100);
            }
            v.sort();

            for _ in 0..20 {
                seed = next_u32(&mut seed);
                let key = (seed % 200) as i32 - 100;
                assert_eq!(lower_bound_i32(&v, key), baseline_lower_bound(&v, key));
            }
        }
    }
}
```

1.6 A Reusable Algorithm Implementation Template

Consistency makes your codebase maintainable. Use a standard template for every algorithm module:

- contract and preconditions at the top,
- invariants near loops,
- complexity and failure modes,
- implementation,
- tests (deterministic + optional randomized),
- performance notes (when relevant).

Template skeleton

```
/* ============================
   Algorithm: <NAME>
   ============================

   Contract:
   - Inputs:
   - Outputs:
   - Preconditions:
   - Postconditions:

   Invariants:
   - (loop invariants)

   Complexity:
   - Time:
   - Memory:

   Failure modes:
   - Panics?
   - Overflow?
   - Recursion depth?

   Notes:
   - Implementation choices, tradeoffs
*/

pub fn algo_name(/* params */) /* -> return */ {
    unimplemented!()
}

#[cfg(test)]
```

```rust
mod tests {
    use super::algo_name;

    #[test]
    fn basic_cases() {
        let _ = algo_name;
    }
}
```

Filled example: Two Sum (HashMap)

```rust
use std::collections::HashMap;

/* ============================
   Algorithm: two_sum
   ============================

   Contract:
   - Input: slice a, target
   - Output: Option<(i,j)> with i<j and a[i]+a[j]==target

   Complexity:
   - Time: O(n) average
   - Memory: O(n)
*/

pub fn two_sum(a: &[i32], target: i32) -> Option<(usize, usize)> {
    let mut pos: HashMap<i32, usize> = HashMap::new();
    for (i, &x) in a.iter().enumerate() {
        let need = target - x;
        if let Some(&j) = pos.get(&need) {
            return Some((j, i));
```

```
        }
        pos.insert(x, i);
    }
    None
}

#[cfg(test)]
mod tests {
    use super::two_sum;

    #[test]
    fn finds_pair() {
        let a = [2, 7, 11, 15];
        assert_eq!(two_sum(&a, 9), Some((0, 1)));
    }

    #[test]
    fn none_when_missing() {
        let a = [1, 2, 3];
        assert_eq!(two_sum(&a, 100), None);
    }

    #[test]
    fn duplicates_work() {
        let a = [3, 3];
        assert_eq!(two_sum(&a, 6), Some((0, 1)));
    }
}
```

1.7 Chapter Conclusion

In this chapter you established the practical foundation for engineering algorithms in Rust:

- Use Cargo as your engineering system: library-first structure, strict tests, and release-mode runs for real performance.
- Let ownership guide design: take `&[T]` for read-only, `&mut [T]` for in-place, and avoid `clone` unless necessary.
- Use `Vec<T>` effectively: pre-allocate capacity, prefer slice APIs, and structure loops to reduce indexing complexity.
- Apply standard collections confidently: `HashMap` for indexing and memoization, `BinaryHeap` for priority-based workflows and top-k problems.
- Write clean tests: deterministic edge cases first, then randomized checks against a baseline when correctness is subtle.
- Follow a reusable template so every algorithm implementation stays consistent, reviewable, and safe to extend.

With these disciplines in place, the next chapters will focus on core algorithm families and build them into a coherent, production-quality toolbox.

Complexity and Performance Discipline

2.1 Time Complexity in Practice

Time complexity is not merely asymptotic notation. In real engineering, it is a combination of:

- Algorithmic order ($O(n)$, $O(n \log n)$, $O(n^2)$, etc.)
- Constant factors
- Memory access patterns
- Branch predictability
- Cache locality
- Compiler optimizations in release mode

In Rust, these dimensions are visible and controllable.

Asymptotics vs Constants

Two algorithms may both be $O(n)$, yet differ significantly in performance because:

- One performs bounds checks in a tight loop,

- One allocates repeatedly,
- One suffers from poor cache locality,
- One uses unnecessary cloning.

Example: naive vs optimized accumulation.

```
/* Naive style: indexing inside loop */
pub fn sum_indexing(a: &[i64]) -> i64 {
    let mut s = 0;
    for i in 0..a.len() {
        s += a[i];
    }
    s
}

/* Iterator style: often cleaner and equally optimized */
pub fn sum_iterator(a: &[i64]) -> i64 {
    a.iter().copied().sum()
}
```

Both are $O(n)$. In optimized builds, LLVM typically eliminates redundant checks, making them similar in performance. However, poorly structured loops can prevent such optimizations.

Nested Loops and Real Cost

Consider duplicate detection:

```
/* O(n^2) duplicate detection */
pub fn has_duplicate_naive(a: &[i32]) -> bool {
    for i in 0..a.len() {
        for j in (i + 1)..a.len() {
            if a[i] == a[j] {
```

```
                return true;
            }
        }
    }
    false
}
```

Versus a HashMap-based solution:

```
use std::collections::HashSet;

/* Average O(n) */
pub fn has_duplicate_hash(a: &[i32]) -> bool {
    let mut seen = HashSet::with_capacity(a.len());
    for &x in a {
        if !seen.insert(x) {
            return true;
        }
    }
    false
}
```

Asymptotically, $O(n)$ beats $O(n^2)$. In practice, for very small inputs, the naive version may be faster due to lower overhead. Engineering discipline requires measuring rather than assuming.

Algorithmic Scaling Demonstration

```
pub fn quadratic_example(n: usize) -> usize {
    let mut c = 0;
    for i in 0..n {
        for j in 0..n {
            if i == j {
                c += 1;
```

```
            }
        }
    }
    c
}

pub fn linear_example(n: usize) -> usize {
    let mut c = 0;
    for i in 0..n {
        if i % 2 == 0 {
            c += 1;
        }
    }
    c
}
```

As n grows, the quadratic function becomes impractical rapidly. This scaling behavior must guide algorithm selection.

2.2 Space Complexity and Memory Behavior

Space complexity matters as much as time complexity. Memory behavior influences:

- Cache efficiency,
- Page faults,
- Allocation overhead,
- Fragmentation,
- Scalability.

Stack Allocation

Stack allocation is:

- Extremely fast,
- Automatically freed,
- Limited in size.

```rust
pub fn stack_array_example() -> i32 {
    let arr = [1, 2, 3, 4, 5];
    arr.iter().sum()
}
```

Heap Allocation

Heap allocation:

- Is flexible in size,
- Requires allocator involvement,
- Has overhead per allocation.

```rust
pub fn heap_vector_example(n: usize) -> i64 {
    let v: Vec<i64> = vec![0; n];
    v.iter().sum()
}
```

Memory Locality

Contiguous memory (e.g., Vec) improves cache performance.

```
pub fn contiguous_scan(v: &[i64]) -> i64 {
    let mut s = 0;
    for &x in v {
        s += x;
    }
    s
}
```

Linked structures degrade locality:

```
pub struct Node {
    pub value: i64,
    pub next: Option<Box<Node>>,
}
```

Traversal of boxed linked nodes incurs pointer chasing and poor cache behavior.

2.3 Stack vs Heap Considerations

Stack Characteristics

- Fixed-size frames,
- Very low overhead,
- Risk of stack overflow with deep recursion.

Recursive example:

```
pub fn factorial_recursive(n: u64) -> u64 {
    if n <= 1 {
        1
    } else {
        n * factorial_recursive(n - 1)
    }
}
```

Deep recursion may overflow stack.

Iterative alternative:

```
pub fn factorial_iterative(n: u64) -> u64 {
    let mut result = 1;
    for i in 2..=n {
        result *= i;
    }
    result
}
```

Heap Growth Patterns

Repeated allocations inside loops degrade performance.

```
/* Inefficient: repeated allocation */
pub fn inefficient_concat(a: &[&str]) -> String {
    let mut s = String::new();
    for &part in a {
        s = s + part;
    }
    s
}
```

Efficient approach:

```rust
pub fn efficient_concat(a: &[&str]) -> String {
    let total: usize = a.iter().map(|s| s.len()).sum();
    let mut out = String::with_capacity(total);
    for &part in a {
        out.push_str(part);
    }
    out
}
```

2.4 Avoiding Unnecessary Allocations

Allocations are among the most expensive operations in tight loops.

Reuse Buffers

```rust
pub fn reuse_buffer(a: &[i32], buffer: &mut Vec<i32>) {
    buffer.clear();
    for &x in a {
        if x > 0 {
            buffer.push(x);
        }
    }
}
```

Use Slices Instead of Cloning

```rust
pub fn print_slice(a: &[i32]) {
    for &x in a {
        println!("{}", x);
    }
}
```

Prefer Iterators That Avoid Allocation

```rust
pub fn count_even(a: &[i32]) -> usize {
    a.iter().filter(|&&x| x % 2 == 0).count()
}
```

2.5 Measuring Performance in Rust

Never assume performance; measure it.

Release Mode is Mandatory

```
cargo run --release
cargo test --release
```

Debug mode includes extra checks and no aggressive optimization.

Simple Timing with Instant

```rust
use std::time::Instant;

pub fn measure_example() {
    let data: Vec<i32> = (0..1_000_000).collect();
    let start = Instant::now();
    let sum: i32 = data.iter().sum();
    let duration = start.elapsed();
    println!("Sum: {}, Time: {:?}", sum, duration);
}
```

Microbenchmark Pattern

```rust
use std::time::Instant;

pub fn benchmark<F: Fn()>(f: F, iterations: usize) {
    let start = Instant::now();
    for _ in 0..iterations {
        f();
    }
    let elapsed = start.elapsed();
    println!("Elapsed: {:?}", elapsed);
}
```

Avoid Benchmark Pitfalls

- Ensure work is not optimized away,
- Use release mode,
- Repeat multiple iterations,
- Warm up caches,
- Compare against baseline.

2.6 Chapter Conclusion

Performance discipline is not optional in algorithm engineering.
This chapter established:

- Time complexity must be understood both asymptotically and practically.

- Space complexity influences cache behavior and scalability.
- Stack vs heap decisions impact performance and reliability.
- Avoiding unnecessary allocations dramatically improves throughput.
- Measuring performance in release mode is mandatory.

From this point forward, every algorithm in this handbook will follow strict discipline:

- Clear complexity analysis,
- Explicit memory behavior,
- Allocation awareness,
- Measured performance in realistic builds.

Correctness is mandatory. Performance is measurable. Discipline makes both sustainable.

Two Pointers & Sliding Window

3.1 Two Sum (Sorted Array)

When the input is a **sorted array**, the Two Sum problem can be solved in **linear time** using two pointers. This is a classic example of using order to avoid hashing and extra memory.

Problem

Given a sorted slice `a` and an integer `target`, find indices `(i, j)` such that:

$$i < j \quad \text{and} \quad a[i] + a[j] = target$$

Return `None` if no solution exists.

Key idea

- Start with `i = 0` (left) and `j = n-1` (right).
- If sum is too small, move left pointer right to increase sum.
- If sum is too large, move right pointer left to decrease sum.

Implementation (Rust)

```rust
pub fn two_sum_sorted(a: &[i32], target: i32) -> Option<(usize, usize)> {
    if a.len() < 2 { return None; }
    let mut i: usize = 0;
    let mut j: usize = a.len() - 1;

    while i < j {
        let s = a[i] + a[j];
        if s == target {
            return Some((i, j));
        } else if s < target {
            i += 1;
        } else {
            j -= 1;
        }
    }
    None
}

#[cfg(test)]
mod tests_two_sum_sorted {
    use super::two_sum_sorted;

    #[test]
    fn finds_pair() {
        let a = [1, 2, 4, 7, 11, 15];
        assert_eq!(two_sum_sorted(&a, 9), Some((1, 3))); // 2 + 7
    }

    #[test]
    fn none_when_missing() {
        let a = [1, 2, 3, 4];
```

```
        assert_eq!(two_sum_sorted(&a, 100), None);
    }

    #[test]
    fn handles_negatives() {
        let a = [-10, -3, 0, 5, 9];
        assert_eq!(two_sum_sorted(&a, -13), Some((0, 1)));
    }
}
```

Complexity

- Time: $O(n)$
- Space: $O(1)$

3.2 Removing Duplicates In-Place

This pattern appears frequently in sorted arrays and streams: **compress a sorted array in-place** so each value appears once.

Problem

Given a sorted array `a`, remove duplicates in-place and return the new logical length. The front of `a` is modified to contain unique elements.

Invariants

- `write` points to the next position to write a new unique element.
- All elements in `a[0..write]` are unique.

Implementation

```rust
pub fn remove_duplicates_sorted(a: &mut [i32]) -> usize {
    if a.is_empty() { return 0; }
    let mut write: usize = 1;
    for read in 1..a.len() {
        if a[read] != a[read - 1] {
            a[write] = a[read];
            write += 1;
        }
    }
    write
}

#[cfg(test)]
mod tests_remove_dup {
    use super::remove_duplicates_sorted;

    #[test]
    fn empty() {
        let mut a: [i32; 0] = [];
        assert_eq!(remove_duplicates_sorted(&mut a), 0);
    }

    #[test]
    fn all_unique() {
        let mut a = [1, 2, 3];
        let n = remove_duplicates_sorted(&mut a);
        assert_eq!(n, 3);
        assert_eq!(&a[..n], &[1, 2, 3]);
    }

    #[test]
```

```
    fn with_duplicates() {
        let mut a = [1, 1, 2, 2, 2, 3, 3, 4];
        let n = remove_duplicates_sorted(&mut a);
        assert_eq!(n, 4);
        assert_eq!(&a[..n], &[1, 2, 3, 4]);
    }
}
```

Complexity

- Time: $O(n)$
- Space: $O(1)$

3.3 The Sliding Window Technique

Sliding window is two pointers on steroids:

- A **window** `[l, r]` over the input.
- Move `r` to expand and include new elements.
- Move `l` to shrink when constraints are violated.

When to use sliding window

Use it when you need:

- max/min length satisfying a condition,
- max/min sum satisfying a condition (with non-negative values),

- substring/subarray with constraints,
- count of windows meeting criteria.

Core reusable template

```rust
/* Template: sliding window with frequency table (or constraint) */
pub fn sliding_window_template(a: &[u8]) -> usize {
    let mut l: usize = 0;
    let mut best: usize = 0;

    /* state describing window */
    let mut freq = [0u32; 256];
    let mut bad = 0u32; /* number of violated constraints (example) */

    for r in 0..a.len() {
        let x = a[r] as usize;
        freq[x] += 1;
        if freq[x] == 2 {
            bad += 1;
        }

        while bad > 0 {
            let y = a[l] as usize;
            if freq[y] == 2 {
                bad -= 1;
            }
            freq[y] -= 1;
            l += 1;
        }

        let len = r + 1 - l;
        if len > best { best = len; }
```

```
    }

    best
}
```

This template uses:

- O(n) time,
- O(1) extra space when alphabet is fixed (ASCII).

3.4 Longest Substring Without Repeating Characters

This problem is the **canonical sliding-window frequency example**.

Problem

Given a string, return the length of the longest substring without repeating characters.

Engineering decision: bytes vs Unicode

If you treat the input as **bytes**, the algorithm is fast and simple. For full Unicode character-level behavior, you must iterate over `char` or grapheme clusters, which changes indexing costs and representation.
This handbook provides:

- A fast **byte-based** version (common in systems contexts),
- A **Unicode-scalar** version (valid Rust `char` iteration).

Fast ASCII/byte version

```rust
pub fn longest_unique_substring_bytes(s: &str) -> usize {
    let b = s.as_bytes();
    let mut last = [-1i32; 256]; /* last index seen */
    let mut l: usize = 0;
    let mut best: usize = 0;

    for (r, &ch) in b.iter().enumerate() {
        let idx = ch as usize;
        let p = last[idx];
        if p >= 0 {
            let p = p as usize;
            if p >= l {
                l = p + 1;
            }
        }
        last[idx] = r as i32;

        let len = r + 1 - l;
        if len > best { best = len; }
    }
    best
}

#[cfg(test)]
mod tests_longest_bytes {
    use super::longest_unique_substring_bytes;

    #[test]
    fn examples() {
        assert_eq!(longest_unique_substring_bytes("abcabcbb"), 3);
        assert_eq!(longest_unique_substring_bytes("bbbbb"), 1);
```

```
        assert_eq!(longest_unique_substring_bytes("pwwkew"), 3);
        assert_eq!(longest_unique_substring_bytes(""), 0);
    }
}
```

Unicode-scalar version (Rust char iteration)

This version treats each `char` (Unicode scalar value) as a unit. It is correct for `char`-based uniqueness, but not for grapheme clusters.

```
use std::collections::HashMap;

pub fn longest_unique_substring_chars(s: &str) -> usize {
    let mut last: HashMap<char, usize> = HashMap::new();
    let mut l: usize = 0;
    let mut best: usize = 0;

    for (r, ch) in s.chars().enumerate() {
        if let Some(&p) = last.get(&ch) {
            if p >= l {
                l = p + 1;
            }
        }
        last.insert(ch, r);
        let len = r + 1 - l;
        if len > best { best = len; }
    }
    best
}
```

3.5 Maximum Subarray Window

There are multiple “maximum subarray” problems. Here we cover a common sliding-window form:

Problem A: Maximum sum of a fixed-size window

Given array a and window size k, find maximum sum of any contiguous subarray of length k.

Implementation

```rust
pub fn max_sum_fixed_window(a: &[i64], k: usize) -> Option<i64> {
    if k == 0 || a.len() < k { return None; }
    let mut sum: i64 = a[..k].iter().sum();
    let mut best = sum;

    for i in k..a.len() {
        sum += a[i];
        sum -= a[i - k];
        if sum > best { best = sum; }
    }
    Some(best)
}

#[cfg(test)]
mod tests_max_sum_fixed {
    use super::max_sum_fixed_window;

    #[test]
    fn basic() {
        let a = [1, 2, 3, 4, 5];
        assert_eq!(max_sum_fixed_window(&a, 2), Some(9)); // 4+5
```

```
    }

    #[test]
    fn invalid() {
        let a = [1, 2, 3];
        assert_eq!(max_sum_fixed_window(&a, 0), None);
        assert_eq!(max_sum_fixed_window(&a, 4), None);
    }
}
```

Problem B: Maximum sum subarray (Kadane)

Kadane is not sliding-window in the strict constraint sense, but it is a two-pointer-like linear scan with a rolling decision: extend or restart.

```
pub fn max_subarray_kadane(a: &[i64]) -> Option<i64> {
    if a.is_empty() { return None; }
    let mut best = a[0];
    let mut cur = a[0];

    for &x in &a[1..] {
        /* either extend previous subarray or start new at x */
        cur = (cur + x).max(x);
        best = best.max(cur);
    }
    Some(best)
}

#[cfg(test)]
mod tests_kadane {
    use super::max_subarray_kadane;
```

```rust
    #[test]
    fn example() {
        let a = [-2, 1, -3, 4, -1, 2, 1, -5, 4];
        assert_eq!(max_subarray_kadane(&a), Some(6)); // 4 + (-1) + 2 + 1
    }
}
```

3.6 Edge Cases and Optimization Strategies

Two-pointer and sliding-window algorithms fail mostly due to:

1) Empty and tiny inputs

Always handle:

- empty slice,
- single-element slice,
- `k = 0`,
- `k > n`.

2) Off-by-one errors

Typical mistakes:

- using `while l <= r` when `r` can underflow,
- using `len = r - l` instead of `r + 1 - l`,
- forgetting to update best after shrinking.

3) Underflow with `usize`

`usize` underflow is a common bug when decrementing. Example: initialize `j = len - 1` only when `len > 0`.

4) Choose the right state representation

For substring problems:

- fixed alphabet (ASCII) → fixed arrays (fast),
- large domain (Unicode chars) → `HashMap`.

5) Avoid unnecessary allocations

Keep everything in slices. Return indices/lengths when possible. Allocate only when returning an actual substring or subarray is required.

6) Prefer one pass, monotonic updates

A disciplined window:

- increments `r` once each step,
- increments `l` monotonically,
- maintains state with O(1) updates.

7) Testing strategy

Test:

- repeated values,

- all unique values,
- alternating patterns,
- very large input (for performance),
- randomized tests compared to a baseline.

Randomized test for two-sum-sorted

```
fn baseline_two_sum_sorted(a: &[i32], target: i32) -> Option<(usize, usize)> {
    for i in 0..a.len() {
        for j in (i + 1)..a.len() {
            if a[i] + a[j] == target {
                return Some((i, j));
            }
        }
    }
    None
}

#[cfg(test)]
mod randomized_two_sum {
    use super::{two_sum_sorted, baseline_two_sum_sorted};

    fn next_u32(s: &mut u32) -> u32 {
        *s = s.wrapping_mul(1664525).wrapping_add(1013904223);
        *s
    }

    #[test]
    fn matches_baseline() {
        let mut seed = 7u32;
```

```
        for _ in 0..200 {
            seed = next_u32(&mut seed);
            let n = (seed % 32) as usize;

            let mut v: Vec<i32> = Vec::with_capacity(n);
            for _ in 0..n {
                seed = next_u32(&mut seed);
                v.push((seed % 50) as i32 - 25);
            }
            v.sort();

            seed = next_u32(&mut seed);
            let target = (seed % 60) as i32 - 30;

            assert_eq!(
                two_sum_sorted(&v, target),
                baseline_two_sum_sorted(&v, target)
            );
        }
    }
}
```

3.7 Chapter Conclusion

Two pointers and sliding window are not “tricks”; they are disciplined ways to build linear-time algorithms.

This chapter delivered:

- Two Sum on sorted data using O(1) memory and O(n) time.
- In-place duplicate removal using read/write pointer invariants.

- A reusable sliding-window template with monotonic movement.
- Longest substring without repeats (fast byte version + Unicode-char version).
- Maximum subarray window patterns: fixed-size window sum and Kadane for general maximum subarray.
- A practical edge-case checklist and optimization strategy focused on avoiding underflow, minimizing allocations, and validating with randomized baselines.

In the next chapter, we will build on these patterns to implement frequency-based windows, monotonic queues, and advanced window constraints that appear in real systems and interview-grade problems alike.

Prefix Sum & Range Techniques

4.1 Prefix Sum Fundamentals

Prefix sums are among the highest-leverage tools in algorithm engineering. They convert repeated range queries from $O(n)$ per query into $O(1)$ per query after an $O(n)$ preprocessing pass.

Definition

Given an array $a[0..n)$, define the prefix sum array $p[0..n]$ as:

$$p[0] = 0, \quad p[i+1] = p[i] + a[i]$$

Then any range sum $a[l] + \cdots + a[r-1]$ is:

$$\text{sum}(l, r) = p[r] - p[l]$$

Engineering discipline

- Use length $n + 1$ to avoid special cases.
- Use a wider integer type for sums (to reduce overflow risk).
- Treat prefix sums as an API: build once, query many times.

Implementation: build and query

```rust
pub fn prefix_sums_i64(a: &[i64]) -> Vec<i64> {
    let mut p = Vec::with_capacity(a.len() + 1);
    p.push(0);
    for &x in a {
        let next = p[p.len() - 1] + x;
        p.push(next);
    }
    p
}

pub fn range_sum(p: &[i64], l: usize, r: usize) -> i64 {
    /* caller guarantees 0 <= l <= r <= n */
    p[r] - p[l]
}

#[cfg(test)]
mod tests_prefix_basic {
    use super::{prefix_sums_i64, range_sum};

    #[test]
    fn build_and_query() {
        let a = [3, -2, 5, 1];
        let p = prefix_sums_i64(&a);
        assert_eq!(p, vec![0, 3, 1, 6, 7]);

        assert_eq!(range_sum(&p, 0, 4), 7);
        assert_eq!(range_sum(&p, 0, 1), 3);
        assert_eq!(range_sum(&p, 1, 3), 3); /* -2 + 5 */
        assert_eq!(range_sum(&p, 2, 2), 0);
    }
}
```

Complexity

- Build: $O(n)$ time, $O(n)$ memory
- Each query: $O(1)$ time

4.2 Subarray Sum Equals K

This problem is the canonical demonstration that prefix sums become powerful when combined with a hash map.

Problem

Given an integer array, count the number of contiguous subarrays whose sum equals k.

Key observation

Let prefix sums be $p[i] = a[0] + \cdots + a[i-1]$. A subarray $[l, r)$ sums to k iff:

$$p[r] - p[l] = k \quad \Rightarrow \quad p[l] = p[r] - k$$

So for each r, we need to know how many earlier prefix sums equal $(p[r] - k)$.

Implementation

```rust
use std::collections::HashMap;

pub fn count_subarrays_sum_k(a: &[i64], k: i64) -> i64 {
    let mut freq: HashMap<i64, i64> = HashMap::new();
    /* prefix sum 0 occurs once before reading any elements */
    freq.insert(0, 1);
```

```
    let mut p: i64 = 0;
    let mut count: i64 = 0;

    for &x in a {
        p += x;
        if let Some(&c) = freq.get(&(p - k)) {
            count += c;
        }
        *freq.entry(p).or_insert(0) += 1;
    }
    count
}

#[cfg(test)]
mod tests_subarray_sum_k {
    use super::count_subarrays_sum_k;

    #[test]
    fn example1() {
        let a = [1, 1, 1];
        assert_eq!(count_subarrays_sum_k(&a, 2), 2);
    }

    #[test]
    fn negatives_supported() {
        let a = [3, 4, -7, 1, 3, 3, 1, -4];
        assert_eq!(count_subarrays_sum_k(&a, 7), 4);
    }

    #[test]
    fn zeros() {
        let a = [0, 0, 0];
        assert_eq!(count_subarrays_sum_k(&a, 0), 6);
```

```
    }
}
```

Complexity

- Time: $O(n)$ average
- Space: $O(n)$ worst-case (distinct prefix sums)

Engineering notes

- Unlike sliding window, this supports negative numbers.
- Use a 64-bit sum even if input is 32-bit.
- The map counts how many times each prefix sum appeared.

4.3 Difference Array Technique

Difference arrays convert a series of **range update operations** into $O(1)$ updates each, followed by a single reconstruction pass.

Problem style

Given n initially-zero values and many operations:

$$\text{add } v \text{ to all } i \in [l, r] \text{ (inclusive)}$$

We want the final array efficiently.

Idea

Maintain a difference array $d[0..n]$ such that:

$$a[i] = d[0] + d[1] + \cdots + d[i]$$

To add v over $[l, r]$:

$$d[l] += v, \quad d[r+1] -= v \quad (\text{if } r+1 < n)$$

Implementation: range add and finalize

```rust
pub fn diff_range_add(n: usize, updates: &[(usize, usize, i64)]) -> Vec<i64> {
    let mut d = vec![0i64; n + 1]; /* n+1 to safely handle r+1 */
    for &(l, r, v) in updates {
        assert!(l <= r);
        assert!(r < n);
        d[l] += v;
        d[r + 1] -= v;
    }

    let mut a = vec![0i64; n];
    let mut cur = 0i64;
    for i in 0..n {
        cur += d[i];
        a[i] = cur;
    }
    a
}

#[cfg(test)]
mod tests_diff {
    use super::diff_range_add;
```

```
    #[test]
    fn basic_range_add() {
        let updates = [
            (1usize, 3usize, 5i64),  /* +5 on [1..3] */
            (0usize, 1usize, 2i64),  /* +2 on [0..1] */
            (2usize, 2usize, -3i64), /* -3 on [2..2] */
        ];
        let a = diff_range_add(5, &updates);
        assert_eq!(a, vec![2, 7, 2, 5, 0]);
    }
}
```

Complexity

- Updates: $O(m)$ for m operations
- Finalize: $O(n)$
- Space: $O(n)$

Where this technique appears

- scheduling and load increments,
- range-based counters,
- offline processing of interval adds,
- efficient simulation of many operations.

4.4 Compact 2D Prefix Sum

2D prefix sums make sum queries over rectangles constant time.

Definition

Given grid g of size $h \times w$, define p of size $(h+1) \times (w+1)$:

$$p[y+1][x+1] = g[y][x] + p[y][x+1] + p[y+1][x] - p[y][x]$$

Then sum of rectangle with corners:

$$(y_0, x_0) \text{ to } (y_1, x_1) \quad \text{(half-open)}$$

is:

$$S = p[y_1][x_1] - p[y_0][x_1] - p[y_1][x_0] + p[y_0][x_0]$$

Engineering decision: store in 1D Vec

For performance and simplicity, store 2D arrays in a single contiguous vector. This avoids nested allocations and improves cache locality.

Implementation

```
pub struct Prefix2D {
    h: usize,
    w: usize,
    p: Vec<i64>, /* (h+1)*(w+1) */
}

impl Prefix2D {
    fn idx(&self, y: usize, x: usize) -> usize {
        y * (self.w + 1) + x
    }

    pub fn new(g: &[Vec<i64>]) -> Self {
        let h = g.len();
```

```
        let w = if h == 0 { 0 } else { g[0].len() };
        for row in g {
            assert_eq!(row.len(), w);
        }

        let mut p = vec![0i64; (h + 1) * (w + 1)];
        let obj = Self { h, w, p };

        let mut out = obj;
        for y in 0..h {
            for x in 0..w {
                let val = g[y][x];
                let a = out.p[out.idx(y, x + 1)];
                let b = out.p[out.idx(y + 1, x)];
                let c = out.p[out.idx(y, x)];
                out.p[out.idx(y + 1, x + 1)] = val + a + b - c;
            }
        }
        out
    }

    pub fn rect_sum(&self, y0: usize, x0: usize, y1: usize, x1: usize) -> i64 {
        assert!(y0 <= y1 && x0 <= x1);
        assert!(y1 <= self.h && x1 <= self.w);

        let p11 = self.p[self.idx(y1, x1)];
        let p01 = self.p[self.idx(y0, x1)];
        let p10 = self.p[self.idx(y1, x0)];
        let p00 = self.p[self.idx(y0, x0)];
        p11 - p01 - p10 + p00
    }
}
```

```rust
#[cfg(test)]
mod tests_prefix2d {
    use super::Prefix2D;

    #[test]
    fn rect_queries() {
        let g = vec![
            vec![1, 2, 3],
            vec![4, 5, 6],
            vec![7, 8, 9],
        ];
        let p = Prefix2D::new(&g);

        assert_eq!(p.rect_sum(0, 0, 3, 3), 45);
        assert_eq!(p.rect_sum(0, 0, 1, 1), 1);
        assert_eq!(p.rect_sum(1, 1, 3, 3), 5 + 6 + 8 + 9);
        assert_eq!(p.rect_sum(0, 1, 2, 3), 2 + 3 + 5 + 6);
    }
}
```

Complexity

- Build: $O(hw)$
- Each query: $O(1)$
- Space: $O(hw)$

4.5 Practical Applications

Prefix sums and range techniques appear constantly in real systems:

1) Fast metrics and counters

If you collect events over time (per second / per minute), prefix sums allow:

- queries like “sum in last 5 minutes” quickly,
- multiple overlapping windows efficiently.

2) Detecting patterns via prefix differences

Many “balance” problems map to prefix sums:

- count of 0 vs 1 in a subarray,
- equal number of two categories,
- longest range satisfying a constraint.

Example: longest subarray with equal 0 and 1

Transform 0 to -1; then the problem becomes longest subarray with sum 0.

```rust
use std::collections::HashMap;

pub fn longest_equal_0_1(bits: &[u8]) -> usize {
    let mut first: HashMap<i64, usize> = HashMap::new();
    first.insert(0, 0);

    let mut p: i64 = 0;
    let mut best: usize = 0;

    for (i, &b) in bits.iter().enumerate() {
        p += if b == 0 { -1 } else { 1 };
        let pos = i + 1;
```

```rust
        if let Some(&start) = first.get(&p) {
            let len = pos - start;
            if len > best { best = len; }
        } else {
            first.insert(p, pos);
        }
    }
    best
}

#[cfg(test)]
mod tests_equal_01 {
    use super::longest_equal_0_1;

    #[test]
    fn example() {
        let bits = [0, 1, 0, 0, 1, 1, 0];
        assert_eq!(longest_equal_0_1(&bits), 6);
    }
}
```

3) Offline interval processing

Difference arrays are used when you must apply many range updates efficiently and only need the final result once.

4) 2D analytics and image-like grids

2D prefix sums power:

- heatmap region queries,

- occupancy grids,
- summed-area tables for fast rectangle sums.

4.6 Chapter Conclusion

This chapter introduced prefix sums and range techniques as practical engineering tools:

- Prefix sums turn repeated range queries into constant time after a linear preprocessing step.
- Combining prefix sums with a hash map yields linear-time solutions to many subarray counting problems, including negatives.
- Difference arrays transform many range updates into O(1) operations each, followed by a single reconstruction pass.
- 2D prefix sums generalize the idea to grids, enabling constant-time rectangle queries with a contiguous-memory implementation.
- Real-world applications include metrics, balancing problems, offline interval updates, and fast grid analytics.

From here, the handbook will build more range-based tools (monotonic queues, interval methods, and heap-based optimizations) using the same discipline: explicit invariants, careful memory behavior, and strong tests.

Binary Search Mastery

5.1 Classic Binary Search

Binary search is the most important example of an algorithm whose correctness depends on a precise contract and a loop invariant. In practice, bugs come from:

- unclear meaning of `lo` and `hi`,
- mixing inclusive/exclusive boundaries,
- failing to prove termination,
- overflow when computing the midpoint,
- not defining behavior for duplicates.

This chapter uses one disciplined model throughout:

$$\texttt{lo} \in [0, n], \quad \texttt{hi} \in [0, n], \quad \text{search range is } [\texttt{lo}, \texttt{hi})$$

Half-open ranges make proofs and code simpler.

Contract (membership search)

Given a sorted slice `a` and a key `key`, return `Some(index)` if the key exists, otherwise `None`. If duplicates exist, we do not promise which occurrence for classic membership search; for deterministic duplicates handling use lower/upper bound.

Loop invariant

At every iteration:

- All candidate indices are in `[lo, hi)`.
- The interval shrinks strictly until `lo == hi`.

Implementation

```rust
pub fn binary_search<T: Ord>(a: &[T], key: &T) -> Option<usize> {
    let mut lo: usize = 0;
    let mut hi: usize = a.len(); /* exclusive */

    while lo < hi {
        /* safe midpoint: avoids overflow */
        let mid = lo + (hi - lo) / 2;
        if &a[mid] < key {
            lo = mid + 1;
        } else if &a[mid] > key {
            hi = mid;
        } else {
            return Some(mid);
        }
    }
    None
}
```

```rust
#[cfg(test)]
mod tests_binary_search {
    use super::binary_search;

    #[test]
    fn empty() {
        let a: [i32; 0] = [];
        assert_eq!(binary_search(&a, &5), None);
    }

    #[test]
    fn found_and_not_found() {
        let a = [1, 3, 5, 7, 9];
        assert_eq!(binary_search(&a, &1), Some(0));
        assert_eq!(binary_search(&a, &9), Some(4));
        assert_eq!(binary_search(&a, &6), None);
    }

    #[test]
    fn duplicates_membership() {
        let a = [2, 2, 2, 2];
        let r = binary_search(&a, &2);
        assert!(matches!(r, Some(i) if i < a.len()));
    }
}
```

Complexity

- Time: $O(\log n)$
- Space: $O(1)$

5.2 Lower Bound and Upper Bound

In real engineering, duplicates exist. You often need deterministic boundary behavior:

- **Lower bound**: first index `i` such that `a[i] >= key`.
- **Upper bound**: first index `i` such that `a[i] > key`.

These definitions produce stable and composable range queries.

Lower bound

```rust
pub fn lower_bound<T: Ord>(a: &[T], key: &T) -> usize {
    let mut lo: usize = 0;
    let mut hi: usize = a.len(); /* exclusive */
    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if &a[mid] < key {
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    lo
}
```

Upper bound

```rust
pub fn upper_bound<T: Ord>(a: &[T], key: &T) -> usize {
    let mut lo: usize = 0;
    let mut hi: usize = a.len(); /* exclusive */
    while lo < hi {
```

```
        let mid = lo + (hi - lo) / 2;
        if &a[mid] <= key {
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    lo
}
```

Using bounds to count duplicates

Count how many times key occurs:

```
pub fn count_equal<T: Ord>(a: &[T], key: &T) -> usize {
    let lb = lower_bound(a, key);
    let ub = upper_bound(a, key);
    ub - lb
}

#[cfg(test)]
mod tests_bounds {
    use super::{lower_bound, upper_bound, count_equal};

    #[test]
    fn basic_bounds() {
        let a = [1, 2, 2, 2, 5, 9];
        assert_eq!(lower_bound(&a, &0), 0);
        assert_eq!(lower_bound(&a, &2), 1);
        assert_eq!(upper_bound(&a, &2), 4);
        assert_eq!(count_equal(&a, &2), 3);
```

```
        assert_eq!(lower_bound(&a, &10), a.len());
        assert_eq!(upper_bound(&a, &10), a.len());
    }
}
```

Engineering note: half-open range is the key

Returning an index in `[0..=n]` makes composition easy:

- empty range when `lb == ub`,
- insertion point is always `lb`.

5.3 Binary Search on the Answer

Many problems do not ask "where is a key in a sorted array?" They ask:

> What is the minimal value x such that a property $P(x)$ becomes true?

If $P(x)$ is monotonic (false, false, ..., true, true), then we can binary-search x.

Core pattern: first true

We search over an integer domain `[lo, hi]` and return the smallest `x` for which `pred(x)` is true.

```
pub fn first_true(mut lo: i64, mut hi: i64, pred: impl Fn(i64) -> bool) -> i64 {
    /* precondition: there exists an answer in [lo, hi] */
    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if pred(mid) {
            hi = mid;
        } else {
```

```
            lo = mid + 1;
        }
    }
    lo
}
```

Example: minimum capacity to ship within D days

Given weights and days, find the minimum ship capacity. Property: if capacity works, larger capacity also works.

```
pub fn can_ship(weights: &[i64], days: i64, cap: i64) -> bool {
    let mut used_days = 1i64;
    let mut cur = 0i64;

    for &w in weights {
        if w > cap { return false; }
        if cur + w <= cap {
            cur += w;
        } else {
            used_days += 1;
            cur = w;
            if used_days > days { return false; }
        }
    }
    true
}

pub fn min_ship_capacity(weights: &[i64], days: i64) -> i64 {
    let mut lo = 0i64;
    let mut hi = 0i64;
    for &w in weights {
```

```
        if w > lo { lo = w; }
        hi += w;
    }
    first_true(lo, hi, |cap| can_ship(weights, days, cap))
}

#[cfg(test)]
mod tests_ship_capacity {
    use super::min_ship_capacity;

    #[test]
    fn example() {
        let w = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
        assert_eq!(min_ship_capacity(&w, 5), 15);
    }
}
```

Engineering checklist for answer-search

- Define the domain bounds tightly (`lo` and `hi`).
- Prove monotonicity of the predicate.
- Ensure `pred(hi)` is true (answer exists).
- Use `first_true` to avoid off-by-one errors.

5.4 Search in a Rotated Array

A rotated sorted array has two sorted halves with a pivot. Binary search still works by identifying which half is sorted at each step.

Problem

Given a rotated sorted array with distinct values, find `key`.

Implementation

```rust
pub fn search_rotated_distinct(a: &[i32], key: i32) -> Option<usize> {
    if a.is_empty() { return None; }
    let mut lo: usize = 0;
    let mut hi: usize = a.len() - 1; /* inclusive */

    while lo <= hi {
        let mid = lo + (hi - lo) / 2;
        if a[mid] == key {
            return Some(mid);
        }

        /* Determine which half is sorted */
        if a[lo] <= a[mid] {
            /* left half sorted */
            if a[lo] <= key && key < a[mid] {
                if mid == 0 { break; }
                hi = mid - 1;
            } else {
                lo = mid + 1;
            }
        } else {
            /* right half sorted */
            if a[mid] < key && key <= a[hi] {
                lo = mid + 1;
            } else {
                if mid == 0 { break; }
                hi = mid - 1;
```

```
            }
        }
    }
    None
}

#[cfg(test)]
mod tests_rotated {
    use super::search_rotated_distinct;

    #[test]
    fn example() {
        let a = [4, 5, 6, 7, 0, 1, 2];
        assert_eq!(search_rotated_distinct(&a, 0), Some(4));
        assert_eq!(search_rotated_distinct(&a, 3), None);
        assert_eq!(search_rotated_distinct(&a, 7), Some(3));
    }

    #[test]
    fn small_cases() {
        let a = [1];
        assert_eq!(search_rotated_distinct(&a, 1), Some(0));
        assert_eq!(search_rotated_distinct(&a, 2), None);
    }
}
```

Engineering note: duplicates make it harder

With duplicates, the sorted-half test becomes ambiguous when `a[lo] == a[mid] == a[hi]`. Then you may need to shrink boundaries linearly in the worst case. This chapter focuses on the distinct-value case for a clean invariant.

5.5 Avoiding Off-by-One and Overflow Errors

Binary search bugs are mostly boundary bugs. This section provides the discipline rules you should apply every time.

Rule 1: choose one boundary model and never mix

Use one of:

- half-open `[lo, hi)` for index search and bounds,
- inclusive `[lo, hi]` only when necessary.

In this handbook:

- membership search and bounds use `[lo, hi)`,
- rotated-array example uses inclusive indices to illustrate that model, but requires extra care.

Rule 2: compute midpoint safely

Never write `mid = (lo + hi)/2` with potentially large integers. Use:

```
mid = lo + (hi - lo)/2
```

Rule 3: enforce progress and termination

For `[lo, hi)` loops:

- when moving right: `lo = mid + 1`
- when moving left: `hi = mid`

This guarantees the interval size decreases.

Rule 4: define duplicate behavior explicitly

- Membership search returns any match.
- Lower bound returns first >=.
- Upper bound returns first >.

Rule 5: avoid usize underflow

When using inclusive boundaries and subtracting 1, guard against `mid == 0`. Prefer half-open ranges when possible.

Diagnostic tests for boundaries

```
#[cfg(test)]
mod boundary_tests {
    use super::{lower_bound, upper_bound, binary_search};

    #[test]
    fn boundaries_and_insertion_points() {
        let a = [10, 20, 20, 30];

        /* insertion at beginning */
        assert_eq!(lower_bound(&a, &5), 0);
        assert_eq!(upper_bound(&a, &5), 0);

        /* inside duplicates */
        assert_eq!(lower_bound(&a, &20), 1);
        assert_eq!(upper_bound(&a, &20), 3);

        /* insertion at end */
        assert_eq!(lower_bound(&a, &40), a.len());
```

```
        assert_eq!(upper_bound(&a, &40), a.len());

        /* membership */
        assert!(binary_search(&a, &20).is_some());
        assert_eq!(binary_search(&a, &25), None);
    }
}
```

5.6 Chapter Conclusion

Binary search is simple in idea and unforgiving in implementation. This chapter provided a disciplined approach that scales:

- Classic membership search using half-open `[lo, hi)` ranges.
- Lower and upper bounds as deterministic tools for duplicates, insertion, and counting.
- Binary search on the answer using the `first_true` monotonic predicate pattern.
- Rotated-array search by identifying the sorted half at each step.
- A strict checklist to eliminate off-by-one, overflow, and `usize` underflow bugs.

In the next chapter, we will extend this discipline to sorting, partitioning, and selection algorithms, where invariants and boundary control matter just as much as they do here.

Sorting and Selection

6.1 `sort` vs `sort_unstable`

Rust provides two primary in-place sorting methods on slices:

- `sort()` and `sort_by(...)`: **stable** sorting
- `sort_unstable()` and `sort_unstable_by(...)`: **unstable** sorting

Both are comparison-based and have the same big-O guarantee:

- Time: $O(n \log n)$ average/worst (comparison sort)
- Space: typically $O(1)$ to $O(n)$ depending on algorithmic strategy for stability

What stable means in practice

Stability preserves the relative order of equal keys.
If we sort records by a key and multiple records share the same key:

- stable sort preserves original order among those equal-key records,
- unstable sort may reorder them arbitrarily.

Engineering decision

- Choose `sort` when you need stable multi-pass sorting (e.g., sort by secondary then primary key).
- Choose `sort_unstable` when stability is not required and you want typically lower overhead and better constant factors.

Demonstration: stability

```rust
#[derive(Clone, Debug, PartialEq, Eq)]
struct Item {
    key: i32,
    id: i32, /* original position marker */
}

pub fn stable_vs_unstable_demo() -> (Vec<Item>, Vec<Item>) {
    let mut a = vec![
        Item { key: 1, id: 10 },
        Item { key: 2, id: 20 },
        Item { key: 1, id: 11 },
        Item { key: 2, id: 21 },
        Item { key: 1, id: 12 },
    ];

    let mut b = a.clone();

    a.sort_by_key(|x| x.key);          /* stable */
    b.sort_unstable_by_key(|x| x.key); /* unstable */

    (a, b)
}
```

```rust
#[cfg(test)]
mod tests_stability {
    use super::{stable_vs_unstable_demo, Item};

    #[test]
    fn stable_preserves_relative_order_for_equal_keys() {
        let (stable, _unstable) = stable_vs_unstable_demo();

        let ones: Vec<i32> = stable.iter().filter(|x| x.key == 1).map(|x| x.id).collect();
        assert_eq!(ones, vec![10, 11, 12]);
    }
}
```

When instability is safe

If your elements are unique under the comparator, stability does not matter. If you only care about the set of values (not original order), stability does not matter.

6.2 Custom Comparators

Real engineering sorting is rarely "natural order." You often sort:

- by a field,
- by multiple keys,
- by computed score,
- by custom partial order (careful).

Rust provides comparator-based methods:

- `sort_by(|a,b| ...)`
- `sort_by_key(|x| ...)`
- unstable equivalents with `sort_unstable_by...`

Comparator contract

Your comparator must define a **total order**:

- transitive,
- antisymmetric,
- consistent (no random decisions).

Violating this can lead to panics or incorrect results.

Sort by multiple keys

```rust
#[derive(Clone, Debug, PartialEq, Eq)]
struct Person {
    last: &'static str,
    first: &'static str,
    age: u32,
}

pub fn sort_people(a: &mut [Person]) {
    a.sort_by(|p, q| {
        p.last.cmp(q.last)
            .then(p.first.cmp(q.first))
            .then(p.age.cmp(&q.age))
    });
}
```

```
#[cfg(test)]
mod tests_multi_key {
    use super::{sort_people, Person};

    #[test]
    fn multi_key_sort() {
        let mut v = vec![
            Person { last: "Z", first: "A", age: 40 },
            Person { last: "A", first: "B", age: 30 },
            Person { last: "A", first: "A", age: 50 },
            Person { last: "A", first: "A", age: 20 },
        ];
        sort_people(&mut v);
        assert_eq!(v[0].first, "A");
        assert_eq!(v[0].age, 20);
        assert_eq!(v[1].age, 50);
        assert_eq!(v[2].first, "B");
    }
}
```

Sort by key extraction

Key extraction is concise and often faster than a full comparator.

```
pub fn sort_by_abs(a: &mut [i32]) {
    a.sort_by_key(|x| x.abs());
}
```

Sorting floats safely

Floating-point values do not implement `Ord` because NaN breaks total ordering. If your domain guarantees no NaN, you can use `partial_cmp(...).unwrap()` carefully.

```rust
pub fn sort_f64_no_nan(a: &mut [f64]) {
    a.sort_by(|x, y| x.partial_cmp(y).unwrap());
}
```

Engineering rule: do this only when you have a strict domain guarantee.

6.3 Quickselect (k-th Element)

Sorting is $O(n \log n)$. Selection can be done in expected $O(n)$ using **Quickselect**:

- Partition around a pivot,
- Recurse only into the side containing the k-th element.

Contract

Given k (0-based), rearrange the slice so that:

- element at index k is the element that would appear there in sorted order,
- elements left of k are <= it,
- elements right of k are >= it,
- full ordering is not guaranteed.

Implementation (iterative, in-place)

We implement a deterministic pivot choice (middle element) for simplicity. In adversarial data, worst-case is $O(n^2)$, but average is $O(n)$. For production hardening, randomized pivot is preferred.

```rust
pub fn quickselect_kth(a: &mut [i32], k: usize) -> i32 {
    assert!(k < a.len());

    let mut lo = 0usize;
    let mut hi = a.len(); /* exclusive */

    while hi - lo > 1 {
        let pivot_index = lo + (hi - lo) / 2;
        let p = partition_around(a, lo, hi, pivot_index);

        if k == p {
            return a[p];
        } else if k < p {
            hi = p;
        } else {
            lo = p + 1;
        }
    }
    a[lo]
}

fn partition_around(a: &mut [i32], lo: usize, hi: usize, pivot_index: usize) -> usize {
    /* partition a[lo..hi) around pivot, return final pivot position */
    a.swap(pivot_index, hi - 1);
    let pivot = a[hi - 1];

    let mut store = lo;
    for i in lo..(hi - 1) {
        if a[i] < pivot {
            a.swap(store, i);
            store += 1;
        }
    }
```

```
    a.swap(store, hi - 1);
    store
}

#[cfg(test)]
mod tests_quickselect {
    use super::quickselect_kth;

    #[test]
    fn kth_matches_sorted() {
        let mut a = [9, 1, 8, 2, 7, 3, 6, 4, 5];
        for k in 0..a.len() {
            let mut b = a;
            let x = quickselect_kth(&mut b, k);
            let mut s = a.to_vec();
            s.sort();
            assert_eq!(x, s[k]);
        }
    }
}
```

Engineering note: selection is a partial order

Quickselect is ideal when you only need:

- median,
- k-th percentile,
- threshold,
- partitioning for further processing.

It avoids the full cost of sorting.

6.4 Top-K Elements Using `BinaryHeap`

Top-k is a frequent requirement: largest k items, smallest k items, k best scores, etc.
Two standard strategies:

- Max-heap: push all then pop k times ($O(n + k \log n)$).
- Fixed-size min-heap of size k ($O(n \log k)$) for large n and small k.

Strategy A: push all, pop k (simple)

```rust
use std::collections::BinaryHeap;

pub fn top_k_simple(a: &[i32], k: usize) -> Vec<i32> {
    let mut heap = BinaryHeap::new();
    for &x in a {
        heap.push(x);
    }
    let k = k.min(heap.len());
    (0..k).filter_map(|_| heap.pop()).collect()
}
```

Strategy B: fixed-size min-heap (efficient for small k)

We keep the k largest seen so far. Maintain a min-heap of size k:

- If heap has less than k items, push.
- Else, if new item is larger than smallest (heap top), replace.

Rust `BinaryHeap` is a max-heap, so we use `Reverse` to simulate a min-heap.

```rust
use std::cmp::Reverse;
use std::collections::BinaryHeap;

pub fn top_k_streaming(a: &[i32], k: usize) -> Vec<i32> {
    if k == 0 { return vec![]; }
    let mut heap: BinaryHeap<Reverse<i32>> = BinaryHeap::new();

    for &x in a {
        if heap.len() < k {
            heap.push(Reverse(x));
        } else if let Some(&Reverse(min_k)) = heap.peek() {
            if x > min_k {
                heap.pop();
                heap.push(Reverse(x));
            }
        }
    }

    let mut out: Vec<i32> = heap.into_iter().map(|r| r.0).collect();
    out.sort_unstable_by(|x, y| y.cmp(x)); /* return descending */
    out
}

#[cfg(test)]
mod tests_topk {
    use super::{top_k_simple, top_k_streaming};

    #[test]
    fn topk_correctness() {
        let a = [5, 1, 9, 2, 9, 3, 7];
        let mut s = a.to_vec();
        s.sort_unstable_by(|x, y| y.cmp(x));
```

```
        assert_eq!(top_k_simple(&a, 3), s[..3].to_vec());
        assert_eq!(top_k_streaming(&a, 3), s[..3].to_vec());
    }

    #[test]
    fn handles_k_edge_cases() {
        let a = [1, 2, 3];
        assert_eq!(top_k_streaming(&a, 0), Vec::<i32>::new());
        assert_eq!(top_k_streaming(&a, 10), vec![3, 2, 1]);
    }
}
```

6.5 Stability vs Performance Tradeoffs

Sorting is never only about big-O. The decision is about contracts and costs:

When stability matters

- Multi-key sorting via multiple passes (secondary then primary).
- Sorting records while preserving original order among equal keys.
- Deterministic outputs for equal keys required by tests or auditing.

When instability is preferred

- Keys are unique or you do not care about equal-key order.
- Performance matters more than stable ordering.
- You want lower overhead and typically better cache behavior.

Selection vs sorting tradeoff

- If you need only a threshold (median, k-th), use selection (Quickselect).
- If you need top-k and k is small compared to n, use heap of size k.
- If you need full ordering, sort is simplest and often optimized heavily.

Choose the right tool

- Full ranking: `sort` / `sort_unstable`
- Median / percentile: Quickselect
- Top-k: heap-based

6.6 Chapter Conclusion

This chapter built practical mastery over sorting and selection:

- You learned the engineering difference between `sort` (stable) and `sort_unstable` (unstable).
- You built custom comparators safely using `cmp().then(...)` patterns and key extraction.
- You implemented Quickselect to find the k-th element in expected linear time without fully sorting.
- You implemented top-k using `BinaryHeap` with two strategies: full heap and fixed-size streaming heap.
- You established clear tradeoffs between stability, performance, and partial ordering needs.

From here, the handbook can move into hash-based algorithms, graph foundations, and dynamic programming patterns with the same discipline: explicit contracts, careful memory behavior, and strong tests.

Hashing Patterns

7.1 Frequency Counting

Frequency counting is the most common hashing pattern:

- count occurrences,
- detect duplicates,
- compute histograms,
- build lookup tables for later queries.

Core idea

Use a map from key $\rightarrow$ count, and update counts in one pass.

Counting integers

```rust
use std::collections::HashMap;

pub fn freq_i32(a: &[i32]) -> HashMap<i32, usize> {
    let mut m: HashMap<i32, usize> = HashMap::with_capacity(a.len());
    for &x in a {
```

```
        *m.entry(x).or_insert(0) += 1;
    }
    m
}

#[cfg(test)]
mod tests_freq_i32 {
    use super::freq_i32;

    #[test]
    fn counts_correctly() {
        let m = freq_i32(&[3, 3, 1, 2, 3, 2]);
        assert_eq!(m.get(&1).copied(), Some(1));
        assert_eq!(m.get(&2).copied(), Some(2));
        assert_eq!(m.get(&3).copied(), Some(3));
    }
}
```

Counting words (String keys)

When keys are strings, prefer minimizing allocations:

- for transient counting inside a function, you can count by `&str` tied to the input lifetime,
- for returning the map beyond the input lifetime, you must allocate owned `String`.

Counting by `&str` (no string allocations)

```
use std::collections::HashMap;

pub fn freq_str<'a>(words: &'a [&'a str]) -> HashMap<&'a str, usize> {
    let mut m: HashMap<&'a str, usize> = HashMap::with_capacity(words.len());
```

```
    for &w in words {
        *m.entry(w).or_insert(0) += 1;
    }
    m
}

#[cfg(test)]
mod tests_freq_str {
    use super::freq_str;

    #[test]
    fn counts_words() {
        let w = ["a", "bb", "a", "a", "bb"];
        let m = freq_str(&w);
        assert_eq!(m.get("a").copied(), Some(3));
        assert_eq!(m.get("bb").copied(), Some(2));
    }
}
```

Counting chars (fixed alphabet optimization)

If the domain is small (ASCII), replace HashMap with a fixed array for speed.

```
pub fn freq_ascii_bytes(s: &str) -> [u32; 256] {
    let mut f = [0u32; 256];
    for &b in s.as_bytes() {
        f[b as usize] += 1;
    }
    f
}

#[cfg(test)]
```

```
mod tests_freq_ascii {
    use super::freq_ascii_bytes;

    #[test]
    fn ascii_count() {
        let f = freq_ascii_bytes("abca");
        assert_eq!(f[b'a' as usize], 2);
        assert_eq!(f[b'b' as usize], 1);
        assert_eq!(f[b'c' as usize], 1);
    }
}
```

7.2 Grouping and Aggregation

Grouping is frequency counting generalized:

- build groups by key,
- aggregate values per key (sum, min/max, mean),
- collect indices or records per category.

Group indices by value

```
use std::collections::HashMap;

pub fn group_indices(a: &[i32]) -> HashMap<i32, Vec<usize>> {
    let mut g: HashMap<i32, Vec<usize>> = HashMap::new();
    for (i, &x) in a.iter().enumerate() {
        g.entry(x).or_insert_with(Vec::new).push(i);
    }
    g
```

```
}

#[cfg(test)]
mod tests_group_indices {
    use super::group_indices;

    #[test]
    fn groups_positions() {
        let g = group_indices(&[5, 1, 5, 2, 1]);
        assert_eq!(g.get(&5).unwrap(), &vec![0, 2]);
        assert_eq!(g.get(&1).unwrap(), &vec![1, 4]);
        assert_eq!(g.get(&2).unwrap(), &vec![3]);
    }
}
```

Aggregate sums by key

```
use std::collections::HashMap;

pub fn sum_by_category(pairs: &[(i32, i64)]) -> HashMap<i32, i64> {
    let mut m: HashMap<i32, i64> = HashMap::with_capacity(pairs.len());
    for &(cat, value) in pairs {
        *m.entry(cat).or_insert(0) += value;
    }
    m
}

#[cfg(test)]
mod tests_sum_by_category {
    use super::sum_by_category;

    #[test]
```

```
    fn sums_correctly() {
        let pairs = [(1, 10), (2, 5), (1, -3), (2, 20)];
        let m = sum_by_category(&pairs);
        assert_eq!(m.get(&1).copied(), Some(7));
        assert_eq!(m.get(&2).copied(), Some(25));
    }
}
```

Group strings by normalized form (anagram buckets)

Classic real-world pattern: group by canonical signature. Here: sort characters to get a stable signature.

```
use std::collections::HashMap;

pub fn group_anagrams(words: &[&str]) -> HashMap<String, Vec<String>> {
    let mut m: HashMap<String, Vec<String>> = HashMap::new();

    for &w in words {
        let mut chars: Vec<char> = w.chars().collect();
        chars.sort_unstable();
        let key: String = chars.into_iter().collect();

        m.entry(key).or_insert_with(Vec::new).push(w.to_string());
    }

    m
}

#[cfg(test)]
mod tests_anagrams {
    use super::group_anagrams;
```

```
    #[test]
    fn groups() {
        let g = group_anagrams(&["eat", "tea", "tan", "ate", "nat", "bat"]);
        assert!(g.values().any(|v| v.len() == 3)); /* eat/tea/ate */
        assert!(g.values().any(|v| v.len() == 2)); /* tan/nat */
        assert!(g.values().any(|v| v.len() == 1)); /* bat */
    }
}
```

7.3 Sliding Window with `HashMap`

When the domain is large (Unicode chars, arbitrary tokens, ids), you cannot use a fixed array. A HashMap-based window maintains counts in the current range.

Pattern: maintain window counts

Window invariant:

- `l` and `r` move forward monotonically,
- map stores counts for elements in `[l, r]`,
- shrinking removes counts, deleting keys at zero to control memory.

Example: longest substring with at most K distinct chars

This is the canonical “HashMap sliding window” problem.

```
use std::collections::HashMap;
```

```rust
pub fn longest_at_most_k_distinct(s: &str, k: usize) -> usize {
    if k == 0 { return 0; }

    let chars: Vec<char> = s.chars().collect(); /* for indexing by usize */
    let mut cnt: HashMap<char, usize> = HashMap::new();

    let mut l: usize = 0;
    let mut distinct: usize = 0;
    let mut best: usize = 0;

    for r in 0..chars.len() {
        let ch = chars[r];
        let entry = cnt.entry(ch).or_insert(0);
        if *entry == 0 {
            distinct += 1;
        }
        *entry += 1;

        while distinct > k {
            let left = chars[l];
            if let Some(v) = cnt.get_mut(&left) {
                *v -= 1;
                if *v == 0 {
                    distinct -= 1;
                    /* keep map smaller */
                    cnt.remove(&left);
                }
            }
            l += 1;
        }

        let len = r + 1 - l;
        if len > best { best = len; }
```

```
    }

    best
}

#[cfg(test)]
mod tests_k_distinct {
    use super::longest_at_most_k_distinct;

    #[test]
    fn basic() {
        assert_eq!(longest_at_most_k_distinct("eceba", 2), 3); /* "ece" */
        assert_eq!(longest_at_most_k_distinct("aa", 1), 2);
        assert_eq!(longest_at_most_k_distinct("", 2), 0);
        assert_eq!(longest_at_most_k_distinct("abc", 0), 0);
    }
}
```

Engineering note: indexing Unicode

Rust `&str` cannot be indexed by integer because UTF-8 is variable width. For window logic that needs integer indexing, you must choose:

- work on bytes (fast, ASCII-oriented),
- collect `chars()` into a Vec<char> (simple, extra allocation),
- or maintain an iterator-based window (more complex).

The above code chooses `Vec<char>` for clarity and correctness.

7.4 When to Use `BTreeMap`

`HashMap` gives average $O(1)$ operations but does not preserve order. `BTreeMap` provides ordered keys with $O(\log n)$ operations.

Use BTreeMap when you need ordering

Common situations:

- output must be sorted by key,
- you need range queries (keys between A and B),
- you need predecessor/successor logic,
- you need stable iteration order for deterministic results.

Example: frequency count with ordered output

```rust
use std::collections::BTreeMap;

pub fn freq_ordered(a: &[i32]) -> BTreeMap<i32, usize> {
    let mut m: BTreeMap<i32, usize> = BTreeMap::new();
    for &x in a {
        *m.entry(x).or_insert(0) += 1;
    }
    m
}

#[cfg(test)]
mod tests_btree_freq {
    use super::freq_ordered;
```

```
    #[test]
    fn ordered_keys() {
        let m = freq_ordered(&[3, 1, 2, 3, 2]);
        let keys: Vec<i32> = m.keys().copied().collect();
        assert_eq!(keys, vec![1, 2, 3]);
    }
}
```

Example: range queries on keys

```
use std::collections::BTreeMap;

pub fn sum_values_in_key_range(m: &BTreeMap<i32, i64>, lo: i32, hi: i32) -> i64 {
    /* sums values for keys in [lo, hi] inclusive */
    m.range(lo..=hi).map(|(_k, v)| *v).sum()
}

#[cfg(test)]
mod tests_btree_range {
    use super::sum_values_in_key_range;
    use std::collections::BTreeMap;

    #[test]
    fn range_sum() {
        let mut m = BTreeMap::new();
        m.insert(1, 10);
        m.insert(3, 30);
        m.insert(5, 50);
        assert_eq!(sum_values_in_key_range(&m, 2, 5), 80);
    }
}
```

Engineering tradeoff

- `HashMap`: faster average lookups, no order, sensitive to hashing overhead.
- `BTreeMap`: ordered, predictable iteration, supports `range`, but $O(\log n)$.

7.5 Recognizing Real-World Patterns

Hashing patterns appear everywhere. Training your eye to recognize them is more valuable than memorizing any single problem.

Pattern 1: count then decide

- Validate constraints: "no element occurs more than k times"
- Detect anomalies: "first element that appears twice"
- Compute majority-like information

Pattern 2: build an index for fast lookups

- Map id $\rightarrow$ record pointer/index
- Map string $\rightarrow$ token id
- Map prefix sum $\rightarrow$ count (subarray techniques)

Pattern 3: grouping

- Group by category, then aggregate
- Normalize then group (anagrams, canonical forms)

Pattern 4: window state

- Maintain counts of items in a moving window
- Shrink when constraint violated
- Expand for each new element

Pattern 5: ordered map needs

- Need smallest/largest key quickly
- Need range queries
- Need deterministic key order

A checklist for choosing the right structure

- Is the key domain small and fixed? Use arrays.
- Do you need average constant time? Use HashMap.
- Do you need ordering or range queries? Use BTreeMap.
- Do you need top-k or priorities? Use BinaryHeap.

7.6 Chapter Conclusion

This chapter established hashing as a set of reusable engineering patterns:

- Frequency counting with `entry` is the base pattern.
- Grouping extends counting into aggregation and bucketing.

- Sliding windows use HashMap when the domain is large or unknown.
- `BTreeMap` is the correct choice when you need sorted keys, deterministic iteration, or range queries.
- Real-world recognition patterns let you map new problems to known solutions quickly: count, index, group, window, or range.

In the next chapter, we will build monotonic structures and queue-based techniques that complement hashing in range problems and streaming analytics.

Stack & Monotonic Stack

8.1 Stack Fundamentals

A stack is a last-in-first-out (LIFO) structure. In algorithm engineering it appears in three primary roles:

- **State machine support**: parsing, expression evaluation, matching parentheses.
- **Depth-first processes**: DFS traversal, backtracking, manual recursion elimination.
- **Monotonic reasoning**: tracking the nearest greater/smaller elements, building range boundaries.

In Rust, the most practical stack implementation is `Vec<T>`:

- `push` = push on stack,
- `pop` = pop from stack,
- `last` = peek.

A minimal stack API (Vec-backed)

```
pub struct Stack<T> {
    v: Vec<T>,
```

```rust
}

impl<T> Stack<T> {
    pub fn new() -> Self { Self { v: Vec::new() } }
    pub fn with_capacity(cap: usize) -> Self { Self { v: Vec::with_capacity(cap) } }

    pub fn push(&mut self, x: T) { self.v.push(x); }
    pub fn pop(&mut self) -> Option<T> { self.v.pop() }
    pub fn peek(&self) -> Option<&T> { self.v.last() }

    pub fn len(&self) -> usize { self.v.len() }
    pub fn is_empty(&self) -> bool { self.v.is_empty() }
}
```

Practical example: balanced parentheses

This problem demonstrates the core stack usage: match an opening token with a closing token.

```rust
pub fn is_balanced_parentheses(s: &str) -> bool {
    let mut st: Vec<char> = Vec::new();
    for ch in s.chars() {
        match ch {
            '(' | '[' | '{' => st.push(ch),
            ')' | ']' | '}' => {
                let open = match st.pop() {
                    Some(x) => x,
                    None => return false,
                };
                let ok = match (open, ch) {
                    ('(', ')') => true,
                    ('[', ']') => true,
                    ('{', '}') => true,
```

```
                    _ => false,
                };
                if !ok { return false; }
            }
            _ => {}
        }
    }
    st.is_empty()
}

#[cfg(test)]
mod tests_balance {
    use super::is_balanced_parentheses;

    #[test]
    fn examples() {
        assert!(is_balanced_parentheses("()"));
        assert!(is_balanced_parentheses("([]{})"));
        assert!(!is_balanced_parentheses("(]"));
        assert!(!is_balanced_parentheses("(()"));
        assert!(!is_balanced_parentheses("])"));
    }
}
```

Why Vec is usually best

- contiguous memory, cache-friendly,
- amortized O(1) push/pop,
- simple ownership model,
- no per-node heap allocation like linked stacks.

8.2 Next Greater Element

Next Greater Element (NGE) is the canonical monotonic stack problem.

Problem

For each position i, find the nearest index $j > i$ such that:

$$a[j] > a[i]$$

If none exists, output `None` (or -1).

Monotonic stack idea

Maintain a stack of indices whose values form a **strictly decreasing** sequence. When you see a new value `x`:

- while the top of stack has smaller value, `x` is its next greater,
- pop and set result,
- push current index.

Each index is pushed once and popped once $\Rightarrow O(n)$.

Implementation: next greater to the right

```rust
pub fn next_greater_right(a: &[i32]) -> Vec<Option<usize>> {
    let n = a.len();
    let mut res = vec![None; n];
    let mut st: Vec<usize> = Vec::with_capacity(n); /* stack of indices */

    for i in 0..n {
```

```
        while let Some(&j) = st.last() {
            if a[i] > a[j] {
                st.pop();
                res[j] = Some(i);
            } else {
                break;
            }
        }
        st.push(i);
    }
    res
}

#[cfg(test)]
mod tests_nge {
    use super::next_greater_right;

    #[test]
    fn basic() {
        let a = [2, 1, 2, 4, 3];
        let r = next_greater_right(&a);
        assert_eq!(r, vec![Some(3), Some(2), Some(3), None, None]);
        /* 2 -> 4 (idx3), 1 -> 2 (idx2), 2 -> 4 (idx3) */
    }

    #[test]
    fn decreasing() {
        let a = [5, 4, 3, 2, 1];
        let r = next_greater_right(&a);
        assert!(r.iter().all(|x| x.is_none()));
    }
}
```

Variant: next greater value instead of index

```rust
pub fn next_greater_value(a: &[i32]) -> Vec<Option<i32>> {
    let idx = next_greater_right(a);
    idx.into_iter().map(|o| o.map(|j| a[j])).collect()
}
```

Nearest smaller element patterns

The same logic works for:

- next smaller element (change comparison),
- previous greater element (scan from left, but answer is on left),
- previous smaller element.

8.3 Largest Rectangle in Histogram

This is the flagship monotonic stack problem. Given bar heights, find the largest rectangle area.

Problem

Given heights $h[0..n)$, each bar has width 1. Find:

$$\max_{l \le r} \left(\min_{i \in [l,r]} h[i] \right) \cdot (r - l + 1)$$

Key idea: compute maximal width for each bar as the limiting height

For each bar i, find:

- nearest index to the left with height strictly smaller,
- nearest index to the right with height strictly smaller,

Then bar i can extend between those boundaries.
A monotonic increasing stack can compute areas in one pass.

Engineering approach: sentinel trick

Add a trailing 0-height bar to force popping all remaining bars.

Implementation (O(n))

```rust
pub fn largest_rectangle_area(h: &[i64]) -> i64 {
    if h.is_empty() { return 0; }

    let n = h.len();
    let mut st: Vec<usize> = Vec::with_capacity(n + 1); /* indices, increasing heights */
    let mut best: i64 = 0;

    for i in 0..=n {
        let cur = if i == n { 0 } else { h[i] };

        while let Some(&top) = st.last() {
            if (if top == n { 0 } else { h[top] }) > cur {
                let height = h[top];
                st.pop();

                let left_less = st.last().copied();
                let left_boundary = match left_less {
                    Some(j) => j + 1,
                    None => 0,
                };
```

```
                let right_boundary = i;
                let width = (right_boundary - left_boundary) as i64;

                let area = height * width;
                if area > best { best = area; }
            } else {
                break;
            }
        }

        st.push(i);
    }

    best
}

#[cfg(test)]
mod tests_histogram {
    use super::largest_rectangle_area;

    #[test]
    fn example() {
        let h = [2, 1, 5, 6, 2, 3];
        assert_eq!(largest_rectangle_area(&h), 10); /* 5*2 or 2*5? actual max is 10 */
    }

    #[test]
    fn simple_cases() {
        assert_eq!(largest_rectangle_area(&[2]), 2);
        assert_eq!(largest_rectangle_area(&[2, 2]), 4);
        assert_eq!(largest_rectangle_area(&[1, 2, 3, 4, 5]), 9); /* 3*3 */
        assert_eq!(largest_rectangle_area(&[5, 4, 3, 2, 1]), 9); /* 3*3 */
    }
```

```
    #[test]
    fn empty() {
        let h: [i64; 0] = [];
        assert_eq!(largest_rectangle_area(&h), 0);
    }
}
```

Why the stack works

The stack maintains indices with non-decreasing heights. When you see a height `cur` smaller than the top height, the top bar's maximal right boundary is known (current index), and the left boundary becomes the new stack top + 1 after popping.
Each index is pushed once and popped once, so the loop is $O(n)$.

8.4 Identifying Monotonic Patterns

Monotonic stacks feel magical until you learn to recognize the pattern.

A monotonic stack is appropriate when

- You need **nearest greater/smaller** to the left or right.
- You need to find a boundary where the first violation occurs.
- A value becomes invalid only when a new value breaks monotonic order.
- Each element can be resolved once when a stronger element arrives.

Common problem shapes

- Next greater element, next smaller element.
- Daily temperatures (next warmer day).
- Stock span problem (previous greater elements).
- Largest rectangle in histogram.
- Maximal rectangle in binary matrix (build histograms per row).
- Sum of subarray minimums / maximums (count contribution via boundaries).

Engineering checklist

Before choosing a monotonic stack, confirm:

- Can each element be pushed once and popped once?
- Does a future element determine the answer for previous ones?
- Can you define an invariant: stack is increasing or decreasing?
- Do you need strict vs non-strict comparisons for duplicates?

Strict vs non-strict and duplicates

Handling duplicates requires careful choice:

- Using > vs >= changes boundary selection.
- For histogram, common is to pop while `h[top] > cur` (strict).
- If you pop on >=, equal heights collapse differently.

Rule: pick one convention and test on arrays with equal heights.

A quick duplicate stress test

```rust
#[cfg(test)]
mod duplicate_stress {
    use super::{next_greater_right, largest_rectangle_area};

    #[test]
    fn equal_values_behavior() {
        let a = [2, 2, 2, 2];
        let r = next_greater_right(&a);
        assert!(r.iter().all(|x| x.is_none()));

        let h = [2, 2, 2, 2];
        assert_eq!(largest_rectangle_area(&h), 8);
    }
}
```

8.5 Chapter Conclusion

This chapter built a disciplined foundation for stacks and monotonic stacks:

- You used `Vec<T>` as an efficient stack and implemented a balanced-parentheses validator.
- You implemented Next Greater Element in linear time using a decreasing stack of indices.
- You solved the Largest Rectangle in Histogram problem with an increasing monotonic stack and a sentinel technique.
- You learned how to recognize monotonic patterns and how strict vs non-strict comparisons affect duplicates.

These techniques are core for range boundary problems, streaming analytics, and many performance-critical systems tasks. In the next chapter, we will extend monotonic reasoning into monotonic queues for sliding-window maxima/minima and other deque-based optimizations.

Heap & Greedy Algorithms

9.1 Internals of `BinaryHeap`

A binary heap is a complete binary tree stored in a contiguous array. Rust exposes it as `std::collections::BinaryHeap<T>`. It is a **max-heap** by default:

- `peek()` returns the largest element (by `Ord`),
- `pop()` removes the largest,
- `push(x)` inserts an element and restores heap order.

Array representation

For a heap stored in a 0-based array `v`:

- parent index: `(i - 1) / 2`
- left child: `2*i + 1`
- right child: `2*i + 2`

Heap invariant for max-heap:

$$v[\text{parent}(i)] \geq v[i]$$

for all valid children.

Operation costs

- `peek`: $O(1)$
- `push`: $O(\log n)$ (sift-up)
- `pop`: $O(\log n)$ (swap root with last, sift-down)
- `from Vec` (heapify): $O(n)$

Why it is fast in practice

- contiguous memory improves cache locality,
- logarithmic path is short,
- comparisons dominate; avoid expensive comparators in hot heaps.

Educational implementation: minimal max-heap

This implementation illustrates the internals (not a replacement for std). It is useful for understanding and debugging.

```rust
pub struct MaxHeap<T: Ord> {
    v: Vec<T>,
}

impl<T: Ord> MaxHeap<T> {
    pub fn new() -> Self { Self { v: Vec::new() } }
    pub fn with_capacity(cap: usize) -> Self { Self { v: Vec::with_capacity(cap) } }

    pub fn len(&self) -> usize { self.v.len() }
    pub fn is_empty(&self) -> bool { self.v.is_empty() }
```

```
pub fn peek(&self) -> Option<&T> {
    self.v.get(0)
}

pub fn push(&mut self, x: T) {
    self.v.push(x);
    self.sift_up(self.v.len() - 1);
}

pub fn pop(&mut self) -> Option<T> {
    let n = self.v.len();
    if n == 0 { return None; }
    if n == 1 { return self.v.pop(); }

    self.v.swap(0, n - 1);
    let out = self.v.pop();
    self.sift_down(0);
    out
}

fn sift_up(&mut self, mut i: usize) {
    while i > 0 {
        let p = (i - 1) / 2;
        if self.v[p] >= self.v[i] { break; }
        self.v.swap(p, i);
        i = p;
    }
}

fn sift_down(&mut self, mut i: usize) {
    let n = self.v.len();
    loop {
```

```
            let l = 2 * i + 1;
            let r = 2 * i + 2;
            if l >= n { break; }

            let mut best = l;
            if r < n && self.v[r] > self.v[l] {
                best = r;
            }

            if self.v[i] >= self.v[best] { break; }
            self.v.swap(i, best);
            i = best;
        }
    }
}

#[cfg(test)]
mod tests_maxheap {
    use super::MaxHeap;

    #[test]
    fn pushes_and_pops_in_order() {
        let mut h = MaxHeap::new();
        h.push(3);
        h.push(1);
        h.push(5);
        h.push(2);

        assert_eq!(h.pop(), Some(5));
        assert_eq!(h.pop(), Some(3));
        assert_eq!(h.pop(), Some(2));
        assert_eq!(h.pop(), Some(1));
        assert_eq!(h.pop(), None);
```

```
    }
}
```

Space and ownership

A heap owns its elements. If you want to avoid moving large structs repeatedly:

- store lightweight keys,
- store indices into an external array,
- store `Box<T>` or `Arc<T>` when ownership sharing is necessary.

9.2 Custom Ordering

Rust heaps rely on the `Ord` implementation of their element type. To customize ordering, you typically:

- wrap elements in a struct and implement `Ord`,
- use `std::cmp::Reverse<T>` for min-heap behavior,
- store tuples where the first element is the priority key.

Min-heap via `Reverse`

```rust
use std::cmp::Reverse;
use std::collections::BinaryHeap;

pub fn min_heap_demo(values: &[i32]) -> Vec<i32> {
    let mut h: BinaryHeap<Reverse<i32>> = BinaryHeap::new();
    for &x in values {
```

```
        h.push(Reverse(x));
    }
    let mut out = Vec::new();
    while let Some(Reverse(x)) = h.pop() {
        out.push(x);
    }
    out
}

#[cfg(test)]
mod tests_minheap {
    use super::min_heap_demo;

    #[test]
    fn ascending_output() {
        let out = min_heap_demo(&[3, 1, 4, 1, 5]);
        assert_eq!(out, vec![1, 1, 3, 4, 5]);
    }
}
```

Custom priority item

Example: schedule tasks by higher priority, and tie-break by earlier timestamp. We define a struct whose `Ord` compares by these fields.

Important: in Rust, `BinaryHeap` is a max-heap, so `Ord` should make "bigger = higher priority".

```
use std::cmp::Ordering;

#[derive(Clone, Debug, Eq, PartialEq)]
pub struct Task {
    pub priority: i32,
    pub created_at: u64, /* smaller means older */
```

```
    pub id: u64,
}

impl Ord for Task {
    fn cmp(&self, other: &Self) -> Ordering {
        /* higher priority first */
        self.priority.cmp(&other.priority)
            /* older first for equal priority */
            .then_with(|| other.created_at.cmp(&self.created_at))
            /* stable-ish tie-break */
            .then_with(|| other.id.cmp(&self.id))
    }
}

impl PartialOrd for Task {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

#[cfg(test)]
mod tests_task_order {
    use super::Task;
    use std::collections::BinaryHeap;

    #[test]
    fn higher_priority_first_then_older() {
        let mut h = BinaryHeap::new();
        h.push(Task { priority: 5, created_at: 100, id: 1 });
        h.push(Task { priority: 5, created_at:  90, id: 2 }); /* older */
        h.push(Task { priority: 9, created_at: 200, id: 3 });

        let a = h.pop().unwrap();
```

```
        assert_eq!(a.priority, 9);

        let b = h.pop().unwrap();
        assert_eq!(b.id, 2); /* older among priority=5 */
    }
}
```

Avoiding comparator bugs

Your ordering must be a total order:

- consistent,
- transitive,
- no floating NaN without explicit handling.

A broken `Ord` can make heaps misbehave.

9.3 Task Scheduling and Prioritization

Task scheduling is one of the most practical uses of heaps. You continuously pick “the next best task” according to a policy:

- earliest deadline first,
- shortest job first,
- highest priority first,
- best ratio first (profit/cost), etc.

Pattern A: process by earliest deadline

We can model deadlines as a min-heap by using `Reverse(deadline)`.
Example: given tasks with deadline and duration, check if all tasks can be done in time by scheduling in deadline order. This is a feasibility check used in many planning systems.

```
use std::cmp::Reverse;
use std::collections::BinaryHeap;

#[derive(Clone, Debug)]
pub struct Job {
    pub deadline: i64,
    pub duration: i64,
}

/* simple feasibility: run jobs in earliest-deadline order */
pub fn feasible_by_deadline(jobs: &[Job]) -> bool {
    let mut h: BinaryHeap<Reverse<(i64, i64)>> = BinaryHeap::new();
    for j in jobs {
        h.push(Reverse((j.deadline, j.duration)));
    }

    let mut t = 0i64;
    while let Some(Reverse((d, dur))) = h.pop() {
        t += dur;
        if t > d { return false; }
    }
    true
}

#[cfg(test)]
mod tests_deadlines {
    use super::{feasible_by_deadline, Job};
```

```
    #[test]
    fn feasibility() {
        let ok = [
            Job { deadline: 3, duration: 1 },
            Job { deadline: 5, duration: 2 },
            Job { deadline: 6, duration: 2 },
        ];
        assert!(feasible_by_deadline(&ok));

        let bad = [
            Job { deadline: 3, duration: 2 },
            Job { deadline: 4, duration: 3 },
        ];
        assert!(!feasible_by_deadline(&bad));
    }
}
```

Pattern B: dynamic scheduling with a priority queue

Typical system: tasks arrive, you always dispatch highest priority next.

```
use std::collections::BinaryHeap;

pub fn dispatch_order(tasks: &[super::Task]) -> Vec<u64> {
    let mut h: BinaryHeap<super::Task> = BinaryHeap::new();
    for t in tasks {
        h.push(t.clone());
    }

    let mut out = Vec::with_capacity(tasks.len());
    while let Some(t) = h.pop() {
```

```
        out.push(t.id);
    }
    out
}
```

Pattern C: keep top-k active tasks

This is common in telemetry: keep best k events seen so far.

```
use std::cmp::Reverse;
use std::collections::BinaryHeap;

pub fn keep_top_k(values: &[i32], k: usize) -> Vec<i32> {
    if k == 0 { return vec![]; }

    let mut h: BinaryHeap<Reverse<i32>> = BinaryHeap::new(); /* min-heap of size k */
    for &x in values {
        if h.len() < k {
            h.push(Reverse(x));
        } else if let Some(&Reverse(min_k)) = h.peek() {
            if x > min_k {
                h.pop();
                h.push(Reverse(x));
            }
        }
    }

    let mut out: Vec<i32> = h.into_iter().map(|r| r.0).collect();
    out.sort_unstable_by(|a, b| b.cmp(a));
    out
}
```

```rust
#[cfg(test)]
mod tests_keep_top_k {
    use super::keep_top_k;

    #[test]
    fn basic() {
        let a = [5, 1, 9, 2, 9, 3, 7];
        assert_eq!(keep_top_k(&a, 3), vec![9, 9, 7]);
    }
}
```

9.4 Designing Greedy Strategies

Greedy algorithms make the best local choice at each step. They are fast and often elegant, but must be justified.

Greedy discipline checklist

Before committing to greedy, you must identify:

- **Objective**: maximize/minimize what?
- **Choice rule**: what local decision do we take?
- **Proof idea**: exchange argument or cut property.
- **State**: what minimal information must be maintained?

Classic example: interval scheduling (max non-overlapping)

Given intervals, select maximum number of non-overlapping intervals. Greedy rule: always pick the interval with the earliest end time.

```rust
#[derive(Clone, Debug)]
pub struct Interval {
    pub start: i64,
    pub end: i64,
}

pub fn max_non_overlapping(mut v: Vec<Interval>) -> usize {
    v.sort_unstable_by(|a, b| a.end.cmp(&b.end).then(a.start.cmp(&b.start)));

    let mut count = 0usize;
    let mut cur_end: Option<i64> = None;

    for it in v {
        if cur_end.is_none() || it.start >= cur_end.unwrap() {
            count += 1;
            cur_end = Some(it.end);
        }
    }
    count
}

#[cfg(test)]
mod tests_intervals {
    use super::{max_non_overlapping, Interval};

    #[test]
    fn classic() {
        let v = vec![
            Interval { start: 1, end: 3 },
            Interval { start: 2, end: 4 },
            Interval { start: 3, end: 5 },
            Interval { start: 0, end: 6 },
            Interval { start: 5, end: 7 },
```

```
            Interval { start: 8, end: 9 },
        ];
        assert_eq!(max_non_overlapping(v), 4);
    }
}
```

Greedy + heap: schedule maximum number of courses

A common pattern: sort by deadline, then keep the longest durations in a max-heap. If total time exceeds deadline, drop the longest course (greedy repair step).

```
use std::collections::BinaryHeap;

#[derive(Clone, Debug)]
pub struct Course {
    pub duration: i64,
    pub deadline: i64,
}

pub fn max_courses(mut courses: Vec<Course>) -> usize {
    courses.sort_unstable_by(|a, b| a.deadline.cmp(&b.deadline));

    let mut total = 0i64;
    let mut heap: BinaryHeap<i64> = BinaryHeap::new(); /* durations */

    for c in courses {
        total += c.duration;
        heap.push(c.duration);

        if total > c.deadline {
            if let Some(longest) = heap.pop() {
                total -= longest;
```

```
            }
        }
    }

    heap.len()
}

#[cfg(test)]
mod tests_courses {
    use super::{max_courses, Course};

    #[test]
    fn example() {
        let v = vec![
            Course { duration: 100, deadline: 200 },
            Course { duration: 200, deadline: 1300 },
            Course { duration: 1000, deadline: 1250 },
            Course { duration: 2000, deadline: 3200 },
        ];
        assert_eq!(max_courses(v), 3);
    }
}
```

Why this works (engineering proof sketch)

- Sorting by deadline ensures we respect the tightest constraints first.
- When we exceed a deadline, removing the longest duration reduces total time the most.
- This greedy repair step preserves the maximum possible count so far.

9.5 Chapter Conclusion

This chapter established heaps and greedy algorithms as production-grade tools:

- You learned the internal array model of heaps and how push/pop are implemented via sift-up and sift-down.
- You implemented and tested a minimal educational max-heap to understand invariants.
- You built custom ordering using `Ord`, tie-breaking rules, and `Reverse` for min-heap behavior.
- You applied heaps to real task scheduling and prioritization patterns: earliest deadline, dispatch queues, and top-k streaming.
- You learned how to design greedy strategies with a discipline checklist and saw two canonical greedy successes: interval scheduling and heap-assisted deadline scheduling.

The next chapter will extend these ideas into graph algorithms where heaps and greedy reasoning power shortest paths and minimum spanning trees.

Core Dynamic Programming Patterns

10.1 Designing State Transitions

Dynamic Programming (DP) is engineering discipline applied to repeated subproblems. A correct DP solution is built from five explicit design steps:

1) Define the state

A state is the minimal information needed to represent a subproblem. Typical choices:

- `dp[i]`: best answer for prefix ending at i,
- `dp[i][j]`: best answer using i items with capacity j,
- `dp[pos][flag]`: best answer at a position with a constraint mode.

2) Define the transition

A transition connects a state to earlier states:

$$dp[i] = \min / \max / \text{sum over choices}(dp[\text{previous}] + \text{cost})$$

Every DP algorithm is a controlled enumeration of choices.

3) Define base cases

Base cases must cover:

- smallest input size,
- edge cases such as empty arrays,
- initial constraint states.

4) Choose evaluation order

- bottom-up: iterate states in dependency order,
- top-down: recursion + memoization (careful about recursion depth).

5) Validate invariants and complexity

- ensure all needed states are computed before use,
- avoid overflow,
- understand time and memory cost.

A reusable DP template (bottom-up)

```
/* ============================
   DP Template (1D)
   ============================
   State: dp[i] = best answer for prefix i
   Base: dp[0] = ...
   Transition: dp[i] = f(dp[i-1], dp[i-2], ...)
   Order: for i in 1..=n
```

```rust
*/
pub fn dp_template(n: usize) -> i64 {
    if n == 0 { return 0; }

    let mut dp = vec![0i64; n + 1];
    dp[0] = 0;
    dp[1] = 1;

    for i in 2..=n {
        dp[i] = dp[i - 1] + dp[i - 2]; /* example recurrence */
    }
    dp[n]
}
```

When DP is the right tool

DP is a strong candidate when:

- there are overlapping subproblems,
- the solution can be expressed as an optimal combination of smaller solutions,
- greedy choice is not obviously correct,
- recursion explodes without memoization.

10.2 Fibonacci Optimization

Fibonacci is the canonical teaching example, but engineering discipline means:

- avoid naive recursion,
- choose types carefully,

- use constant memory when possible.

Naive recursion (anti-pattern)

```rust
pub fn fib_naive(n: u64) -> u64 {
    if n <= 1 { return n; }
    fib_naive(n - 1) + fib_naive(n - 2)
}
```

This is exponential time and wastes repeated work.

Memoized recursion (top-down)

```rust
pub fn fib_memo(n: usize) -> u64 {
    fn go(i: usize, memo: &mut Vec<Option<u64>>) -> u64 {
        if let Some(v) = memo[i] { return v; }
        let v = if i <= 1 { i as u64 } else { go(i - 1, memo) + go(i - 2, memo) };
        memo[i] = Some(v);
        v
    }

    let mut memo = vec![None; n + 1];
    go(n, &mut memo)
}

#[cfg(test)]
mod tests_fib_memo {
    use super::fib_memo;

    #[test]
    fn small() {
        assert_eq!(fib_memo(0), 0);
        assert_eq!(fib_memo(1), 1);
```

```
        assert_eq!(fib_memo(10), 55);
    }
}
```

Bottom-up DP (iterative)

```
pub fn fib_dp(n: usize) -> u64 {
    if n <= 1 { return n as u64; }
    let mut dp = vec![0u64; n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for i in 2..=n {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    dp[n]
}
```

Constant memory optimization

We only need the last two values.

```
pub fn fib_optimized(n: usize) -> u64 {
    if n <= 1 { return n as u64; }
    let mut a: u64 = 0;
    let mut b: u64 = 1;
    for _ in 2..=n {
        let c = a + b;
        a = b;
        b = c;
    }
    b
```

```
}

#[cfg(test)]
mod tests_fib_opt {
    use super::fib_optimized;

    #[test]
    fn checks() {
        assert_eq!(fib_optimized(2), 1);
        assert_eq!(fib_optimized(3), 2);
        assert_eq!(fib_optimized(10), 55);
    }
}
```

10.3 Climbing Stairs

Problem

Given n steps, you can climb 1 or 2 steps each time. How many distinct ways to reach the top?

State

$$dp[i] = \text{number of ways to reach step } i$$

Transition

$$dp[i] = dp[i-1] + dp[i-2]$$

Implementation (optimized)

```
pub fn climb_stairs(n: usize) -> u64 {
```

```
    if n == 0 { return 1; } /* one way: do nothing */
    if n == 1 { return 1; }

    let mut a: u64 = 1; /* dp[0] */
    let mut b: u64 = 1; /* dp[1] */

    for _ in 2..=n {
        let c = a + b;
        a = b;
        b = c;
    }
    b
}

#[cfg(test)]
mod tests_stairs {
    use super::climb_stairs;

    #[test]
    fn examples() {
        assert_eq!(climb_stairs(0), 1);
        assert_eq!(climb_stairs(1), 1);
        assert_eq!(climb_stairs(2), 2);
        assert_eq!(climb_stairs(3), 3);
        assert_eq!(climb_stairs(4), 5);
    }
}
```

Engineering note

This is Fibonacci shifted by one index. Use it to train your ability to translate a recurrence into code safely.

10.4 House Robber

Problem

Given an array of non-negative integers `a`, each value is money in a house. You cannot rob adjacent houses. Find the maximum money you can rob.

State

$$dp[i] = \text{max money robbing from prefix } a[0..i)$$

So `dp[0]=0`, `dp[1]=a[0]`.

Transition

At house `i-1`, you either:

- skip it: `dp[i-1]`,
- rob it: `dp[i-2] + a[i-1]`.

$$dp[i] = \max(dp[i-1], dp[i-2] + a[i-1])$$

Implementation (O(1) memory)

```rust
pub fn house_robber(a: &[i64]) -> i64 {
    let mut prev2: i64 = 0; /* dp[i-2] */
    let mut prev1: i64 = 0; /* dp[i-1] */

    for &x in a {
        let take = prev2 + x;
        let skip = prev1;
        let cur = if take > skip { take } else { skip };
```

```
        prev2 = prev1;
        prev1 = cur;
    }
    prev1
}

#[cfg(test)]
mod tests_robber {
    use super::house_robber;

    #[test]
    fn examples() {
        assert_eq!(house_robber(&[1, 2, 3, 1]), 4);
        assert_eq!(house_robber(&[2, 7, 9, 3, 1]), 12);
        assert_eq!(house_robber(&[]), 0);
        assert_eq!(house_robber(&[5]), 5);
    }
}
```

Interpretation

This DP is a two-state machine:

- `prev1`: best answer up to previous house,
- `prev2`: best answer up to the house before previous.

Each new house updates the machine.

10.5 Memory Optimization Techniques

DP often starts with a large table. Engineering discipline is to reduce memory once you understand dependencies.

Technique 1: rolling variables (1D)

If `dp[i]` depends only on `dp[i-1]` and `dp[i-2]`, keep two variables. Used in Fibonacci, stairs, robber.

Technique 2: rolling array (k-step dependencies)

If `dp[i]` depends on last `k` states, keep a ring buffer of size `k`.

```rust
/* Example: dp[i] depends on dp[i-1], dp[i-2], ..., dp[i-k] */
pub fn dp_k_rolling(n: usize, k: usize) -> i64 {
    if n == 0 { return 0; }
    let mut ring = vec![0i64; k.max(1)];
    ring[0] = 1;

    for i in 1..=n {
        let mut s = 0i64;
        for t in 1..=k {
            if i >= t {
                s += ring[(i - t) % k];
            }
        }
        ring[i % k] = s;
    }
    ring[n % k]
}
```

Technique 3: reduce dimension (2D $\rightarrow$ 1D)

If `dp[i][j]` depends only on the previous row `dp[i-1][*]`, compress 2D into 1D.
Classic example: 0/1 knapsack (shown here as a pattern only).

```rust
pub fn knapsack_01(values: &[i64], weights: &[usize], cap: usize) -> i64 {
    assert_eq!(values.len(), weights.len());
    let mut dp = vec![0i64; cap + 1];

    for i in 0..values.len() {
        let w = weights[i];
        let v = values[i];

        /* iterate backwards to avoid reusing item i multiple times */
        for c in (w..=cap).rev() {
            let candidate = dp[c - w] + v;
            if candidate > dp[c] {
                dp[c] = candidate;
            }
        }
    }
    dp[cap]
}

#[cfg(test)]
mod tests_knapsack {
    use super::knapsack_01;

    #[test]
    fn small() {
        let values = [6, 10, 12];
        let weights = [1usize, 2usize, 3usize];
        assert_eq!(knapsack_01(&values, &weights, 5), 22);
    }
}
```

Technique 4: choose top-down vs bottom-up consciously

- Top-down memoization can compute only needed states, reducing work.
- Bottom-up avoids recursion depth issues and can be faster and more cache-friendly.

Technique 5: store only what you must return

Sometimes you need only the best value, not the path. Sometimes you need reconstruction; then store parent pointers or decisions.
Engineering rule: do not store reconstruction info unless required.

10.6 Chapter Conclusion

This chapter built the core DP mindset and patterns:

- DP design starts with explicit state definitions and precise transitions, not with code.
- Fibonacci shows the full evolution: naive recursion $\rightarrow$ memoization $\rightarrow$ bottom-up $\rightarrow$ constant-memory.
- Climbing stairs is a direct recurrence-to-code exercise that trains correct base cases.
- House robber introduces the critical DP pattern: choose between "take" and "skip" with rolling state.
- Memory optimization techniques are systematic: rolling variables, rolling buffers, 2D-to-1D compression, and conscious choice of evaluation order.

From here, the handbook can move to more powerful DP families: grid DP, subsequence DP, and DP with monotonic optimizations, all built on the same disciplined state-transition design.

The Knapsack Pattern

11.1 0/1 Knapsack

The 0/1 knapsack pattern models decisions where each item can be taken at most once. It appears across engineering domains:

- feature selection under memory budget,
- packing payloads under size limits,
- choosing tasks under time budget,
- selecting modules under power constraints.

Problem

You have n items. Item i has:

- weight w_i (cost),
- value v_i (benefit).

Given capacity C, choose a subset maximizing total value while total weight $\leq C$.

DP state

$$dp[c] = \text{maximum value achievable with capacity } c$$

We update dp item by item.

Transition

For each item (w, v):

$$dp[c] = \max(dp[c],\ dp[c - w] + v)$$

But for 0/1 knapsack, the key discipline rule is:

> Iterate capacities **backwards** so each item is used at most once.

Implementation (space-optimized 1D)

```rust
pub fn knapsack_01(values: &[i64], weights: &[usize], cap: usize) -> i64 {
    assert_eq!(values.len(), weights.len());

    let mut dp = vec![0i64; cap + 1];

    for i in 0..values.len() {
        let w = weights[i];
        let v = values[i];

        if w > cap { continue; }

        for c in (w..=cap).rev() {
            let cand = dp[c - w] + v;
            if cand > dp[c] {
                dp[c] = cand;
            }
        }
```

```
    }

    dp[cap]
}

#[cfg(test)]
mod tests_knapsack_01 {
    use super::knapsack_01;

    #[test]
    fn classic_small() {
        let values = [6, 10, 12];
        let weights = [1usize, 2usize, 3usize];
        assert_eq!(knapsack_01(&values, &weights, 5), 22);
    }

    #[test]
    fn capacity_tight() {
        let values = [5, 6, 7];
        let weights = [3usize, 4usize, 5usize];
        assert_eq!(knapsack_01(&values, &weights, 2), 0);
        assert_eq!(knapsack_01(&values, &weights, 3), 5);
    }
}
```

Reconstruction (optional)

If you must recover which items were chosen, you need extra state:

- store decisions per item-capacity (2D parent pointers),
- or store predecessor info at each capacity (careful with overwrites).

Engineering rule: do not store reconstruction unless required.

2D reference implementation (clear, heavier memory)

```rust
pub fn knapsack_01_2d(values: &[i64], weights: &[usize], cap: usize) -> i64 {
    assert_eq!(values.len(), weights.len());
    let n = values.len();

    let mut dp = vec![vec![0i64; cap + 1]; n + 1];

    for i in 1..=n {
        let w = weights[i - 1];
        let v = values[i - 1];
        for c in 0..=cap {
            dp[i][c] = dp[i - 1][c]; /* skip */
            if w <= c {
                let cand = dp[i - 1][c - w] + v;
                if cand > dp[i][c] {
                    dp[i][c] = cand; /* take */
                }
            }
        }
    }

    dp[n][cap]
}

#[cfg(test)]
mod tests_knapsack_01_2d {
    use super::knapsack_01_2d;

    #[test]
    fn matches_known() {
        let values = [6, 10, 12];
        let weights = [1usize, 2usize, 3usize];
```

```
        assert_eq!(knapsack_01_2d(&values, &weights, 5), 22);
    }
}
```

Complexity

- Time: $O(n \cdot C)$
- Space: $O(C)$ (1D optimized) or $O(n \cdot C)$ (2D)

11.2 Unbounded Knapsack

Unbounded knapsack allows taking each item multiple times. This changes only one thing:

> Iterate capacities **forwards** to allow reuse of the same item.

Problem

Same as knapsack, but each item can be chosen unlimited times.

Transition

$$dp[c] = \max(dp[c],\ dp[c-w] + v)$$

But now the update order for capacity must be increasing: `for c in w..=cap`.

Implementation

```
pub fn knapsack_unbounded(values: &[i64], weights: &[usize], cap: usize) -> i64 {
    assert_eq!(values.len(), weights.len());
    let mut dp = vec![0i64; cap + 1];
```

```
    for i in 0..values.len() {
        let w = weights[i];
        let v = values[i];
        if w > cap { continue; }

        for c in w..=cap {
            let cand = dp[c - w] + v;
            if cand > dp[c] {
                dp[c] = cand;
            }
        }
    }

    dp[cap]
}

#[cfg(test)]
mod tests_unbounded {
    use super::knapsack_unbounded;

    #[test]
    fn simple() {
        /* choose weight=2 value=3 multiple times */
        let values = [3, 4];
        let weights = [2usize, 3usize];
        assert_eq!(knapsack_unbounded(&values, &weights, 7), 10); /* 2+2+3 */
        assert_eq!(knapsack_unbounded(&values, &weights, 6), 9);  /* 2+2+2 */
    }
}
```

Engineering note: coin change is unbounded knapsack

Many coin-change variants are unbounded knapsack:

- maximize value = maximize reward,
- minimize coins = replace max with min and initialize with infinity.

11.3 Space Optimization

Space optimization is not a trick; it is a dependency analysis.

Why 1D works for knapsack

In 2D form:

$$dp[i][c] = \max(dp[i-1][c],\ dp[i-1][c-w_i] + v_i)$$

So row i depends only on row $i-1$. Hence we can compress to 1D:

- **0/1**: update backwards to prevent reuse,
- **unbounded**: update forwards to allow reuse.

Common failure: wrong iteration direction

Wrong direction changes the problem:

- 0/1 + forward update $\Rightarrow$ accidentally becomes unbounded.
- unbounded + backward update $\Rightarrow$ restricts reuse incorrectly.

Direction test harness

```rust
pub fn knapsack_01_wrong(values: &[i64], weights: &[usize], cap: usize) -> i64 {
    /* WRONG: forward update turns it into unbounded behavior */
    assert_eq!(values.len(), weights.len());
    let mut dp = vec![0i64; cap + 1];
    for i in 0..values.len() {
        let w = weights[i];
        let v = values[i];
        if w > cap { continue; }
        for c in w..=cap {
            let cand = dp[c - w] + v;
            if cand > dp[c] { dp[c] = cand; }
        }
    }
    dp[cap]
}

#[cfg(test)]
mod tests_direction_bug {
    use super::{knapsack_01, knapsack_01_wrong};

    #[test]
    fn shows_the_difference() {
        let values = [10];
        let weights = [2usize];

        /* 0/1: can take at most once */
        assert_eq!(knapsack_01(&values, &weights, 4), 10);

        /* wrong direction allows taking twice */
        assert_eq!(knapsack_01_wrong(&values, &weights, 4), 20);
    }
```

```
}
```

Space/time tradeoff discipline

- 1D DP: best memory, harder reconstruction.
- 2D DP: easier reconstruction and debugging, more memory.
- choose based on requirements, not habit.

11.4 Resource Allocation Case Study

This case study frames knapsack as an engineering decision tool.

Scenario

You maintain a Windows Rust service that can load optional modules. Each module:

- consumes memory budget (MB),
- provides a measurable benefit score (performance/security/features).

You want the maximum benefit under a memory budget.
This is exactly 0/1 knapsack.

Model

- capacity C = allowed memory (MB),
- weights w_i = module memory (MB),
- values v_i = benefit score.

Implementation: choose modules under budget

Below we compute the best score. We also demonstrate optional reconstruction by storing a choice table.

```
#[derive(Clone, Debug)]
pub struct Module {
    pub name: &'static str,
    pub mem_mb: usize,
    pub score: i64,
}

pub fn best_module_score(mods: &[Module], budget_mb: usize) -> i64 {
    let values: Vec<i64> = mods.iter().map(|m| m.score).collect();
    let weights: Vec<usize> = mods.iter().map(|m| m.mem_mb).collect();
    knapsack_01(&values, &weights, budget_mb)
}

/* Reconstruction version: returns chosen module indices */
pub fn choose_modules(mods: &[Module], budget_mb: usize) -> Vec<usize> {
    let n = mods.len();
    let cap = budget_mb;

    let mut dp = vec![vec![0i64; cap + 1]; n + 1];

    for i in 1..=n {
        let w = mods[i - 1].mem_mb;
        let v = mods[i - 1].score;

        for c in 0..=cap {
            dp[i][c] = dp[i - 1][c];
            if w <= c {
                let cand = dp[i - 1][c - w] + v;
```

```
                if cand > dp[i][c] {
                    dp[i][c] = cand;
                }
            }
        }
    }

    /* reconstruct */
    let mut c = cap;
    let mut picked: Vec<usize> = Vec::new();
    for i in (1..=n).rev() {
        if dp[i][c] != dp[i - 1][c] {
            picked.push(i - 1);
            c -= mods[i - 1].mem_mb;
        }
    }
    picked.reverse();
    picked
}

#[cfg(test)]
mod tests_case_study {
    use super::{best_module_score, choose_modules, Module};

    #[test]
    fn module_selection() {
        let mods = [
            Module { name: "parser",   mem_mb: 30, score: 60 },
            Module { name: "crypto",   mem_mb: 40, score: 90 },
            Module { name: "telemetry",mem_mb: 20, score: 30 },
            Module { name: "cache",    mem_mb: 25, score: 50 },
            Module { name: "jit",      mem_mb: 60, score: 120 },
        ];
```

```
        let budget = 90;
        let score = best_module_score(&mods, budget);
        assert_eq!(score, 200); /* crypto(90) + jit(120)? not possible (40+60=100). best is
        ↪  crypto+parser+cache = 90+60+50=200 with mem 40+30+25=95 (too big). let's verify
        ↪  with reconstruction below */
        let picked = choose_modules(&mods, budget);

        let total_mem: usize = picked.iter().map(|&i| mods[i].mem_mb).sum();
        let total_score: i64 = picked.iter().map(|&i| mods[i].score).sum();

        assert!(total_mem <= budget);
        assert_eq!(total_score, score);
    }
}
```

Engineering note about the test

The reconstruction test is the authority:

- it verifies budget constraints,
- it verifies score consistency.

This pattern prevents accidental “expected value” mistakes in unit tests.

Extending the case study

Once you have the knapsack model, you can evolve it:

- multiple budgets (memory + CPU) ⇒ multi-dimensional knapsack,
- minimum score requirement ⇒ feasibility DP,
- per-module dependencies ⇒ graph constraints (more complex).

11.5 Chapter Conclusion

This chapter delivered the knapsack pattern as a reusable engineering tool:

- 0/1 knapsack models one-time choices under a capacity constraint, solved by DP with backward capacity updates.
- unbounded knapsack models repeatable choices, solved by forward capacity updates.
- space optimization is a dependency decision: 2D $\rightarrow$ 1D when each row depends only on the previous.
- update direction is not cosmetic: it determines whether items can be reused.
- a practical resource allocation case study showed how knapsack maps cleanly to real module selection problems, including optional reconstruction of chosen items.

From here, you can recognize knapsack everywhere: budgets, quotas, capacity planning, feature gating, packing, and constrained optimization across systems.

Graph Representation in Rust

12.1 Adjacency List Modeling

Graphs in real systems are almost always sparse. For sparse graphs, the adjacency list is the default representation:

- memory scales with edges, not with V^2,
- traversal is efficient and cache-friendly if stored contiguously,
- easy to attach metadata per edge.

Core model

Let vertices be numbered `0..n`. Adjacency list:

$$g[u] = \{v \mid (u \to v) \in E\}$$

In Rust, the standard representation is:

```rust
pub type Graph = Vec<Vec<usize>>;
```

Building a simple adjacency list (undirected)

```rust
pub fn build_undirected(n: usize, edges: &[(usize, usize)]) -> Vec<Vec<usize>> {
    let mut g = vec![Vec::<usize>::new(); n];
    for &(u, v) in edges {
        assert!(u < n && v < n);
        g[u].push(v);
        g[v].push(u);
    }
    g
}

#[cfg(test)]
mod tests_build_undirected {
    use super::build_undirected;

    #[test]
    fn basic() {
        let edges = [(0, 1), (1, 2), (2, 0)];
        let g = build_undirected(3, &edges);

        assert_eq!(g[0].len(), 2);
        assert!(g[0].contains(&1));
        assert!(g[0].contains(&2));
    }
}
```

Engineering improvements

For performance and correctness in production:

- pre-allocate adjacency vectors using degree counts,

- sort neighbors when deterministic order or binary search is needed,
- optionally deduplicate edges if input may contain duplicates.

Pre-allocation by degree counting

This avoids repeated reallocation inside `push()` loops.

```rust
pub fn build_undirected_prealloc(n: usize, edges: &[(usize, usize)]) -> Vec<Vec<usize>> {
    let mut deg = vec![0usize; n];
    for &(u, v) in edges {
        assert!(u < n && v < n);
        deg[u] += 1;
        deg[v] += 1;
    }

    let mut g: Vec<Vec<usize>> = (0..n).map(|i| Vec::with_capacity(deg[i])).collect();
    for &(u, v) in edges {
        g[u].push(v);
        g[v].push(u);
    }
    g
}

#[cfg(test)]
mod tests_prealloc {
    use super::build_undirected_prealloc;

    #[test]
    fn builds() {
        let edges = [(0, 1), (0, 2), (0, 3)];
        let g = build_undirected_prealloc(4, &edges);
        assert_eq!(g[0].len(), 3);
```

```
    }
}
```

When adjacency lists are not enough

Adjacency lists are perfect for traversal. But if you need:

- constant-time edge existence tests between arbitrary u and v,
- extremely dense graphs,

then adjacency matrices or bitsets may be a better fit. This handbook focuses on sparse-graph engineering.

12.2 Directed vs Undirected Graphs

A graph is **directed** if edges have orientation:

$$u \to v$$

It is **undirected** if each edge connects both ways:

$$u \leftrightarrow v$$

Modeling difference in adjacency lists

- Directed: store only `u -> v` in `g[u]`
- Undirected: store both `u -> v` and `v -> u`

Build directed graph

```rust
pub fn build_directed(n: usize, edges: &[(usize, usize)]) -> Vec<Vec<usize>> {
    let mut g = vec![Vec::<usize>::new(); n];
    for &(u, v) in edges {
        assert!(u < n && v < n);
        g[u].push(v);
    }
    g
}

#[cfg(test)]
mod tests_directed {
    use super::build_directed;

    #[test]
    fn orientation_matters() {
        let edges = [(0, 1)];
        let g = build_directed(2, &edges);
        assert_eq!(g[0], vec![1]);
        assert_eq!(g[1].len(), 0);
    }
}
```

Practical operations affected by direction

- In-degree and out-degree are distinct in directed graphs.
- Reachability differs: u may reach v without v reaching u.
- Many algorithms (topological sort, SCC) require directed modeling.

Compute in-degrees (directed)

```rust
pub fn indegrees(g: &[Vec<usize>]) -> Vec<usize> {
    let n = g.len();
    let mut indeg = vec![0usize; n];
    for u in 0..n {
        for &v in &g[u] {
            indeg[v] += 1;
        }
    }
    indeg
}

#[cfg(test)]
mod tests_indeg {
    use super::{build_directed, indegrees};

    #[test]
    fn basic() {
        let g = build_directed(4, &[(0, 1), (0, 2), (2, 3)]);
        let indeg = indegrees(&g);
        assert_eq!(indeg, vec![0, 1, 1, 1]);
    }
}
```

12.3 Weighted Graph Structures

Weighted graphs attach a weight (cost, distance, latency, capacity) to edges. Common domains:

- routing and shortest paths (latency),
- dependency scheduling (cost),

- resource networks (capacity),
- game AI navigation (distance).

Weighted adjacency list: (to, weight)

The standard representation:

```rust
pub type WGraph = Vec<Vec<(usize, i64)>>;
```

Build weighted directed graph

```rust
pub fn build_weighted_directed(n: usize, edges: &[(usize, usize, i64)]) -> Vec<Vec<(usize, i64)>> {
    let mut g = vec![Vec::<(usize, i64)>::new(); n];
    for &(u, v, w) in edges {
        assert!(u < n && v < n);
        g[u].push((v, w));
    }
    g
}

#[cfg(test)]
mod tests_wdir {
    use super::build_weighted_directed;

    #[test]
    fn builds() {
        let g = build_weighted_directed(3, &[(0, 1, 7), (0, 2, 2), (2, 1, 3)]);
        assert_eq!(g[0].len(), 2);
        assert!(g[0].contains(&(1, 7)));
        assert!(g[0].contains(&(2, 2)));
    }
```

```
}
```

Build weighted undirected graph

```
pub fn build_weighted_undirected(n: usize, edges: &[(usize, usize, i64)]) -> Vec<Vec<(usize,
↪  i64)>> {
    let mut g = vec![Vec::<(usize, i64)>::new(); n];
    for &(u, v, w) in edges {
        assert!(u < n && v < n);
        g[u].push((v, w));
        g[v].push((u, w));
    }
    g
}

#[cfg(test)]
mod tests_wundir {
    use super::build_weighted_undirected;

    #[test]
    fn builds() {
        let g = build_weighted_undirected(3, &[(0, 1, 7), (1, 2, 4)]);
        assert!(g[0].contains(&(1, 7)));
        assert!(g[1].contains(&(0, 7)));
        assert!(g[1].contains(&(2, 4)));
        assert!(g[2].contains(&(1, 4)));
    }
}
```

Engineering discipline: choose the weight type

Weight type depends on domain:

- `i64` for general costs (safe range),
- `u64` for non-negative distances/latencies,
- avoid floating weights in core graph algorithms unless strictly necessary.

Engineering discipline: avoid overflow in path computations

Shortest path algorithms often sum many weights. Use:

- a sentinel for infinity,
- saturating operations if needed,
- domain checks.

A safe infinity pattern for i64 distances

```rust
pub const INF: i64 = i64::MAX / 4;

pub fn relax_example(dist_u: i64, w: i64, dist_v: &mut i64) {
    if dist_u >= INF { return; }
    let cand = dist_u + w;
    if cand < *dist_v {
        *dist_v = cand;
    }
}

#[cfg(test)]
mod tests_inf {
    use super::{INF, relax_example};

    #[test]
    fn relax() {
```

```
        let mut dv = INF;
        relax_example(10, 5, &mut dv);
        assert_eq!(dv, 15);

        let mut dv2 = 100;
        relax_example(INF, 5, &mut dv2);
        assert_eq!(dv2, 100);
    }
}
```

Alternative: edge list for some algorithms

Some algorithms (e.g., Bellman-Ford, Kruskal) work naturally with an edge list:

```
#[derive(Clone, Copy, Debug)]
pub struct Edge {
    pub u: usize,
    pub v: usize,
    pub w: i64,
}
```

You can keep both representations:

- adjacency list for traversal,
- edge list for global edge processing.

Memory layout note

A graph of `Vec<Vec<T>>` contains many heap allocations (one per vertex). For very large graphs, you may consider a compressed representation (CSR-like):

- one big edges array,

- offsets per vertex.

This chapter focuses on the most practical and idiomatic representation first.

Compressed adjacency list (CSR-style, weighted)

This reduces allocations and improves iteration locality.

```rust
#[derive(Clone, Debug)]
pub struct CSRWeighted {
    pub n: usize,
    pub off: Vec<usize>,          /* length n+1 */
    pub to: Vec<usize>,           /* length m */
    pub w: Vec<i64>,              /* length m */
}

impl CSRWeighted {
    pub fn from_edges_directed(n: usize, edges: &[(usize, usize, i64)]) -> Self {
        let mut deg = vec![0usize; n];
        for &(u, v, _w) in edges {
            assert!(u < n && v < n);
            deg[u] += 1;
        }

        let mut off = vec![0usize; n + 1];
        for i in 0..n {
            off[i + 1] = off[i] + deg[i];
        }

        let m = edges.len();
        let mut to = vec![0usize; m];
        let mut w = vec![0i64; m];
        let mut cur = off[..n].to_vec();
```

```rust
        for &(u, v, ww) in edges {
            let idx = cur[u];
            to[idx] = v;
            w[idx] = ww;
            cur[u] += 1;
        }

        Self { n, off, to, w }
    }

    pub fn neighbors(&self, u: usize) -> impl Iterator<Item = (usize, i64)> + '_ {
        let a = self.off[u];
        let b = self.off[u + 1];
        (a..b).map(move |i| (self.to[i], self.w[i]))
    }
}

#[cfg(test)]
mod tests_csr {
    use super::CSRWeighted;

    #[test]
    fn builds_and_iterates() {
        let edges = [(0, 1, 7), (0, 2, 2), (2, 1, 3)];
        let g = CSRWeighted::from_edges_directed(3, &edges);

        let v: Vec<(usize, i64)> = g.neighbors(0).collect();
        assert_eq!(v.len(), 2);
        assert!(v.contains(&(1, 7)));
        assert!(v.contains(&(2, 2)));
    }
}
```

12.4 Chapter Conclusion

This chapter established practical graph representation patterns in Rust:

- Adjacency lists (`Vec<Vec<usize>>`) are the default for sparse graphs, and can be optimized via degree pre-allocation.
- Directed graphs store only forward edges, while undirected graphs store both directions; the difference affects degrees and reachability.
- Weighted graphs extend adjacency lists to store `(to, weight)` pairs and require disciplined choice of numeric types and safe infinity handling.
- For very large graphs, compressed (CSR-style) representations reduce allocations and improve locality, while still enabling efficient neighbor iteration.

With these representations in place, the next chapters can implement traversal and shortest-path algorithms with performance-aware data layouts and precise invariants.

BFS and DFS

13.1 Breadth-First Search

Breadth-First Search (BFS) explores a graph in layers:

- visits all vertices at distance 0 (the start),
- then distance 1 neighbors,
- then distance 2 neighbors, and so on.

For unweighted graphs, BFS computes shortest path lengths in number of edges.

Core structure

BFS requires:

- a queue (FIFO),
- a visited array (or distance array),
- an adjacency list.

In Rust, use `std::collections::VecDeque` for the queue.

BFS: distances from a single source

```rust
use std::collections::VecDeque;

pub fn bfs_distances(g: &[Vec<usize>], start: usize) -> Vec<i32> {
    let n = g.len();
    assert!(start < n);

    let mut dist = vec![-1i32; n];
    let mut q: VecDeque<usize> = VecDeque::new();

    dist[start] = 0;
    q.push_back(start);

    while let Some(u) = q.pop_front() {
        let du = dist[u];
        for &v in &g[u] {
            if dist[v] == -1 {
                dist[v] = du + 1;
                q.push_back(v);
            }
        }
    }

    dist
}

#[cfg(test)]
mod tests_bfs_dist {
    use super::bfs_distances;

    #[test]
    fn distances() {
```

```
        let g = vec![
            vec![1, 2],  /* 0 */
            vec![0, 3],  /* 1 */
            vec![0, 3],  /* 2 */
            vec![1, 2],  /* 3 */
        ];
        assert_eq!(bfs_distances(&g, 0), vec![0, 1, 1, 2]);
        assert_eq!(bfs_distances(&g, 3), vec![2, 1, 1, 0]);
    }
}
```

BFS: shortest path reconstruction

To reconstruct an actual path, store a parent array.

```
use std::collections::VecDeque;

pub fn bfs_shortest_path(g: &[Vec<usize>], start: usize, goal: usize) -> Option<Vec<usize>>
↪ {
    let n = g.len();
    assert!(start < n && goal < n);

    let mut dist = vec![-1i32; n];
    let mut parent: Vec<Option<usize>> = vec![None; n];
    let mut q: VecDeque<usize> = VecDeque::new();

    dist[start] = 0;
    q.push_back(start);

    while let Some(u) = q.pop_front() {
        if u == goal { break; }
        let du = dist[u];
```

```
        for &v in &g[u] {
            if dist[v] == -1 {
                dist[v] = du + 1;
                parent[v] = Some(u);
                q.push_back(v);
            }
        }
    }

    if dist[goal] == -1 { return None; }

    let mut path = Vec::new();
    let mut cur = goal;
    path.push(cur);
    while cur != start {
        cur = parent[cur].unwrap();
        path.push(cur);
    }
    path.reverse();
    Some(path)
}

#[cfg(test)]
mod tests_bfs_path {
    use super::bfs_shortest_path;

    #[test]
    fn path() {
        let g = vec![
            vec![1, 2],
            vec![0, 3],
            vec![0, 3],
            vec![1, 2],
```

```
        ];
        let p = bfs_shortest_path(&g, 0, 3).unwrap();
        assert!(p == vec![0, 1, 3] || p == vec![0, 2, 3]);
    }
}
```

Engineering notes

- Use `dist == -1` as the visited marker in integer-distance BFS.
- For large graphs, adjacency lists should be pre-allocated and possibly stored in CSR form for locality.
- BFS is iterative (stack-safe), unlike recursive DFS.

13.2 Depth-First Search

Depth-First Search (DFS) explores as deep as possible before backtracking. DFS is used for:

- reachability,
- connected components,
- topological sort (directed acyclic graphs),
- cycle detection,
- articulation points and bridges (advanced).

Recursive DFS (simple, but depth risk)

Recursive DFS is concise, but can overflow the call stack if the graph depth is very large. Use iterative DFS for safety in production.

Iterative DFS using a Vec stack

```rust
pub fn dfs_iterative(g: &[Vec<usize>], start: usize) -> Vec<usize> {
    let n = g.len();
    assert!(start < n);

    let mut seen = vec![false; n];
    let mut order = Vec::new();

    let mut st: Vec<usize> = Vec::new();
    st.push(start);

    while let Some(u) = st.pop() {
        if seen[u] { continue; }
        seen[u] = true;
        order.push(u);

        /* push neighbors; reverse for deterministic traversal if needed */
        for &v in g[u].iter().rev() {
            if !seen[v] {
                st.push(v);
            }
        }
    }

    order
}

#[cfg(test)]
mod tests_dfs_iter {
    use super::dfs_iterative;

    #[test]
```

```
    fn visits_reachable() {
        let g = vec![
            vec![1, 2],
            vec![0, 3],
            vec![0],
            vec![1],
        ];
        let order = dfs_iterative(&g, 0);
        assert_eq!(order.len(), 4);
    }
}
```

DFS timestamps (entry/exit times)

Many graph algorithms rely on timestamps:

- discovery time (tin),
- finish time (tout).

Here is an iterative DFS that records discovery order and parent pointers.

```
pub fn dfs_with_parent(g: &[Vec<usize>], start: usize) -> (Vec<Option<usize>>, Vec<usize>) {
    let n = g.len();
    assert!(start < n);

    let mut parent = vec![None; n];
    let mut seen = vec![false; n];
    let mut disc = Vec::new();

    /* stack holds (node, next neighbor index to process) */
    let mut st: Vec<(usize, usize)> = Vec::new();
    st.push((start, 0));
```

```
    parent[start] = None;

    while let Some((u, idx)) = st.pop() {
        if !seen[u] {
            seen[u] = true;
            disc.push(u);
        }

        if idx < g[u].len() {
            let v = g[u][idx];
            /* push current back with next index */
            st.push((u, idx + 1));
            if !seen[v] {
                parent[v] = Some(u);
                st.push((v, 0));
            }
        }
    }

    (parent, disc)
}

#[cfg(test)]
mod tests_parent {
    use super::dfs_with_parent;

    #[test]
    fn parent_tree() {
        let g = vec![
            vec![1, 2],
            vec![0, 3],
            vec![0],
            vec![1],
```

```
        ];
        let (p, disc) = dfs_with_parent(&g, 0);
        assert_eq!(disc[0], 0);
        assert!(p[0].is_none());
        assert!(p[3].is_some());
    }
}
```

13.3 Connected Components

A connected component (in an undirected graph) is a maximal set of vertices where each vertex can reach every other.

Approach

Run BFS or DFS from every unvisited vertex. Each run discovers one component.

Implementation: components with BFS

```
use std::collections::VecDeque;

pub fn connected_components(g: &[Vec<usize>]) -> Vec<Vec<usize>> {
    let n = g.len();
    let mut seen = vec![false; n];
    let mut comps: Vec<Vec<usize>> = Vec::new();

    for s in 0..n {
        if seen[s] { continue; }
        let mut comp = Vec::new();
        let mut q: VecDeque<usize> = VecDeque::new();
```

```
        seen[s] = true;
        q.push_back(s);

        while let Some(u) = q.pop_front() {
            comp.push(u);
            for &v in &g[u] {
                if !seen[v] {
                    seen[v] = true;
                    q.push_back(v);
                }
            }
        }

        comps.push(comp);
    }

    comps
}

#[cfg(test)]
mod tests_components {
    use super::connected_components;

    #[test]
    fn finds_components() {
        let g = vec![
            vec![1],     /* 0 */
            vec![0],     /* 1 */
            vec![3],     /* 2 */
            vec![2],     /* 3 */
            vec![],      /* 4 isolated */
        ];
        let comps = connected_components(&g);
```

```
        assert_eq!(comps.len(), 3);
        assert!(comps.iter().any(|c| c.len() == 2));
        assert!(comps.iter().any(|c| c.len() == 1));
    }
}
```

Engineering notes

- If the graph is directed, the equivalent concept is **strongly connected components** (SCC), which requires different algorithms.
- For very large graphs, store components as IDs per node (`comp_id[u]`) instead of storing explicit lists.

Component IDs (more scalable output)

```
use std::collections::VecDeque;

pub fn component_ids(g: &[Vec<usize>]) -> Vec<usize> {
    let n = g.len();
    let mut seen = vec![false; n];
    let mut id = vec![usize::MAX; n];
    let mut cur_id = 0usize;

    for s in 0..n {
        if seen[s] { continue; }
        let mut q: VecDeque<usize> = VecDeque::new();
        seen[s] = true;
        id[s] = cur_id;
        q.push_back(s);

        while let Some(u) = q.pop_front() {
```

```
            for &v in &g[u] {
                if !seen[v] {
                    seen[v] = true;
                    id[v] = cur_id;
                    q.push_back(v);
                }
            }
        }

        cur_id += 1;
    }

    id
}

#[cfg(test)]
mod tests_comp_ids {
    use super::component_ids;

    #[test]
    fn ids() {
        let g = vec![
            vec![1],
            vec![0],
            vec![3],
            vec![2],
            vec![],
        ];
        let id = component_ids(&g);
        assert_eq!(id[0], id[1]);
        assert_ne!(id[0], id[2]);
        assert_ne!(id[2], id[4]);
    }
```

```
}
```

13.4 Cycle Detection

Cycle detection differs between undirected and directed graphs. Engineering discipline starts by naming which world you are in.

Cycle detection in undirected graphs

In an undirected graph, a cycle exists if during DFS you find an edge to an already visited vertex that is not the parent.

Implementation (DFS iterative)

```rust
pub fn has_cycle_undirected(g: &[Vec<usize>]) -> bool {
    let n = g.len();
    let mut seen = vec![false; n];

    for s in 0..n {
        if seen[s] { continue; }

        let mut st: Vec<(usize, usize)> = Vec::new(); /* (node, parent) */
        st.push((s, usize::MAX));

        while let Some((u, p)) = st.pop() {
            if seen[u] { continue; }
            seen[u] = true;

            for &v in &g[u] {
                if !seen[v] {
                    st.push((v, u));
```

```
                } else if v != p {
                    return true;
                }
            }
        }
    }

    false
}

#[cfg(test)]
mod tests_cycle_undirected {
    use super::has_cycle_undirected;

    #[test]
    fn detects_cycle() {
        let g_cycle = vec![
            vec![1, 2],
            vec![0, 2],
            vec![0, 1],
        ];
        assert!(has_cycle_undirected(&g_cycle));

        let g_tree = vec![
            vec![1],
            vec![0, 2],
            vec![1],
        ];
        assert!(!has_cycle_undirected(&g_tree));
    }
}
```

Cycle detection in directed graphs

In directed graphs, the key concept is the recursion stack (active path). You detect a cycle when you find an edge to a node that is currently "in progress".
We model colors:

- 0 = unvisited,
- 1 = visiting (in recursion stack),
- 2 = done.

Implementation (iterative simulation of recursion)

```rust
pub fn has_cycle_directed(g: &[Vec<usize>]) -> bool {
    let n = g.len();
    let mut color = vec![0u8; n];

    for s in 0..n {
        if color[s] != 0 { continue; }

        /* stack frames: (node, next neighbor index) */
        let mut st: Vec<(usize, usize)> = Vec::new();
        st.push((s, 0));

        while let Some((u, idx)) = st.pop() {
            if color[u] == 0 {
                color[u] = 1; /* enter */
            }

            if idx < g[u].len() {
                let v = g[u][idx];
                /* resume u later */
```

```
                st.push((u, idx + 1));

                if color[v] == 0 {
                    st.push((v, 0));
                } else if color[v] == 1 {
                    return true; /* back edge */
                }
            } else {
                color[u] = 2; /* exit */
            }
        }
    }

    false
}

#[cfg(test)]
mod tests_cycle_directed {
    use super::has_cycle_directed;

    #[test]
    fn detects_directed_cycles() {
        let g_cycle = vec![
            vec![1],
            vec![2],
            vec![0],
        ];
        assert!(has_cycle_directed(&g_cycle));

        let g_dag = vec![
            vec![1, 2],
            vec![3],
            vec![3],
```

```
            vec![],
        ];
        assert!(!has_cycle_directed(&g_dag));
    }
}
```

Engineering notes

- Undirected cycles: parent check is sufficient.
- Directed cycles: need active-path detection (color/state).
- For directed acyclic graphs (DAG), you can topologically sort; a failure indicates a cycle.

13.5 Chapter Conclusion

This chapter delivered BFS and DFS as production-grade building blocks:

- BFS uses `VecDeque` to explore layer-by-layer and compute shortest distances in unweighted graphs, with optional parent pointers for path reconstruction.
- DFS explores depth-first and is best implemented iteratively in Rust for stack safety on large graphs.
- Connected components in undirected graphs are found by running BFS/DFS from each unvisited vertex; scalable outputs use component IDs.
- Cycle detection depends on graph type: parent checks for undirected graphs, and color/state tracking for directed graphs.

With these traversal primitives, the next chapter can build higher-level graph algorithms: topological sorting, shortest paths, and minimum spanning trees.

Shortest Path and Connectivity

14.1 Dijkstra with `BinaryHeap`

Dijkstra's algorithm computes shortest paths from a single source in a graph with **non-negative** edge weights. It is a greedy algorithm that repeatedly finalizes the currently known closest unsettled vertex.

Preconditions

- All edge weights must be ≥ 0.
- Graph can be directed or undirected.
- For negative weights, use Bellman-Ford or specialized algorithms.

Data model

We use a weighted adjacency list:

```rust
pub type WGraph = Vec<Vec<(usize, i64)>>; /* (to, weight) */
```

Why `BinaryHeap` needs a workaround

Rust's `BinaryHeap` is a max-heap. Dijkstra needs a min-priority queue by distance. We use `Reverse((dist, node))`.

Infinity discipline

Distances can overflow if you sum many weights. Use a safe large sentinel:

```
pub const INF: i64 = i64::MAX / 4;
```

Implementation: single-source shortest distances

This is the standard practical version: it allows multiple entries per node in the heap and discards stale ones.

```
use std::cmp::Reverse;
use std::collections::BinaryHeap;

pub fn dijkstra(g: &WGraph, start: usize) -> Vec<i64> {
    let n = g.len();
    assert!(start < n);

    let mut dist = vec![INF; n];
    dist[start] = 0;

    let mut heap: BinaryHeap<Reverse<(i64, usize)>> = BinaryHeap::new();
    heap.push(Reverse((0, start)));

    while let Some(Reverse((d, u))) = heap.pop() {
        if d != dist[u] {
            continue; /* stale entry */
```

```
        }

        for &(v, w) in &g[u] {
            /* non-negative weight requirement */
            debug_assert!(w >= 0);

            if dist[u] >= INF { continue; }
            let nd = d + w;
            if nd < dist[v] {
                dist[v] = nd;
                heap.push(Reverse((nd, v)));
            }
        }
    }

    dist
}

#[cfg(test)]
mod tests_dijkstra {
    use super::{dijkstra, INF, WGraph};

    #[test]
    fn basic_weighted() {
        let g: WGraph = vec![
            vec![(1, 4), (2, 1)],      /* 0 */
            vec![(3, 1)],              /* 1 */
            vec![(1, 2), (3, 5)],      /* 2 */
            vec![],                    /* 3 */
        ];
        let dist = dijkstra(&g, 0);
        assert_eq!(dist[0], 0);
        assert_eq!(dist[2], 1);
```

```
        assert_eq!(dist[1], 3); /* 0->2->1 */
        assert_eq!(dist[3], 4); /* 0->2->1->3 */
        assert!(dist.iter().all(|&x| x <= INF));
    }
}
```

Path reconstruction

To reconstruct paths, store a `parent` for each relaxation.

```
use std::cmp::Reverse;
use std::collections::BinaryHeap;

pub fn dijkstra_with_parent(g: &WGraph, start: usize) -> (Vec<i64>, Vec<Option<usize>>) {
    let n = g.len();
    assert!(start < n);

    let mut dist = vec![INF; n];
    let mut parent: Vec<Option<usize>> = vec![None; n];

    dist[start] = 0;

    let mut heap: BinaryHeap<Reverse<(i64, usize)>> = BinaryHeap::new();
    heap.push(Reverse((0, start)));

    while let Some(Reverse((d, u))) = heap.pop() {
        if d != dist[u] { continue; }

        for &(v, w) in &g[u] {
            debug_assert!(w >= 0);
            let nd = d + w;
            if nd < dist[v] {
```

```
                dist[v] = nd;
                parent[v] = Some(u);
                heap.push(Reverse((nd, v)));
            }
        }
    }

    (dist, parent)
}

pub fn reconstruct_path(parent: &[Option<usize>], start: usize, goal: usize) ->
↪  Option<Vec<usize>> {
    if start == goal { return Some(vec![start]); }
    let mut path = Vec::new();
    let mut cur = goal;

    path.push(cur);
    while cur != start {
        cur = match parent[cur] {
            Some(p) => p,
            None => return None,
        };
        path.push(cur);
    }
    path.reverse();
    Some(path)
}

#[cfg(test)]
mod tests_dijkstra_path {
    use super::{dijkstra_with_parent, reconstruct_path, WGraph};

    #[test]
```

```
    fn reconstructs() {
        let g: WGraph = vec![
            vec![(1, 4), (2, 1)],
            vec![(3, 1)],
            vec![(1, 2), (3, 5)],
            vec![],
        ];
        let (_dist, parent) = dijkstra_with_parent(&g, 0);
        let p = reconstruct_path(&parent, 0, 3).unwrap();
        assert_eq!(p, vec![0, 2, 1, 3]);
    }
}
```

Engineering notes

- Using `d != dist[u]` to skip stale heap entries is simple and fast.
- If you need to detect unreachable nodes, test `dist[v] >= INF`.
- For multi-source shortest paths, initialize the heap with multiple starts at distance 0.

14.2 Disjoint Set (Union-Find)

Union-Find (Disjoint Set Union, DSU) maintains a partition of elements into disjoint sets. It supports:

- `find(x)`: return representative of x's set,
- `union(a, b)`: merge sets,
- `same(a, b)`: connectivity query.

With **path compression** and **union by size/rank**, operations are almost constant time.

Implementation (path compression + union by size)

```rust
#[derive(Clone, Debug)]
pub struct DSU {
    parent: Vec<usize>,
    size: Vec<usize>,
}

impl DSU {
    pub fn new(n: usize) -> Self {
        let mut parent = Vec::with_capacity(n);
        let mut size = Vec::with_capacity(n);
        for i in 0..n {
            parent.push(i);
            size.push(1);
        }
        Self { parent, size }
    }

    pub fn find(&mut self, x: usize) -> usize {
        let p = self.parent[x];
        if p == x {
            x
        } else {
            let r = self.find(p);
            self.parent[x] = r; /* path compression */
            r
        }
    }

    pub fn union(&mut self, a: usize, b: usize) -> bool {
        let mut ra = self.find(a);
        let mut rb = self.find(b);
```

```
        if ra == rb { return false; }

        /* union by size: attach smaller to larger */
        if self.size[ra] < self.size[rb] {
            core::mem::swap(&mut ra, &mut rb);
        }
        self.parent[rb] = ra;
        self.size[ra] += self.size[rb];
        true
    }

    pub fn same(&mut self, a: usize, b: usize) -> bool {
        self.find(a) == self.find(b)
    }

    pub fn set_size(&mut self, x: usize) -> usize {
        let r = self.find(x);
        self.size[r]
    }
}

#[cfg(test)]
mod tests_dsu {
    use super::DSU;

    #[test]
    fn unions_and_queries() {
        let mut d = DSU::new(5);
        assert!(!d.same(0, 1));
        d.union(0, 1);
        d.union(1, 2);
        assert!(d.same(0, 2));
        assert_eq!(d.set_size(2), 3);
```

```
        assert!(!d.same(0, 4));
    }
}
```

Engineering notes

- DSU is ideal for offline connectivity in undirected graphs.
- DSU does not support efficient edge deletions (dynamic connectivity requires different structures).
- DSU works with integer ids; for arbitrary keys, map keys to ids first using `HashMap`.

14.3 Connectivity Problems

Connectivity questions come in multiple forms. This section shows the common patterns and which tool to choose.

Pattern A: are two nodes connected? (undirected, static)

Use DSU or BFS/DFS:

- DSU is best when you have many queries after building edges.
- BFS/DFS is best for a few queries or when you also need traversal info.

Example: answer multiple connectivity queries with DSU

```
pub fn connectivity_queries(n: usize, edges: &[(usize, usize)], queries: &[(usize, usize)])
↪ -> Vec<bool> {
    let mut dsu = super::DSU::new(n);
```

```
    for &(u, v) in edges {
        dsu.union(u, v);
    }
    queries.iter().map(|&(a, b)| dsu.same(a, b)).collect()
}

#[cfg(test)]
mod tests_conn_queries {
    use super::connectivity_queries;

    #[test]
    fn answers() {
        let edges = [(0, 1), (1, 2), (3, 4)];
        let q = [(0, 2), (0, 4), (3, 4)];
        let ans = connectivity_queries(5, &edges, &q);
        assert_eq!(ans, vec![true, false, true]);
    }
}
```

Pattern B: number of connected components

You can compute this via DSU by counting unique roots.

```
use std::collections::HashSet;

pub fn num_components_dsu(n: usize, edges: &[(usize, usize)]) -> usize {
    let mut d = super::DSU::new(n);
    for &(u, v) in edges {
        d.union(u, v);
    }

    let mut roots = HashSet::new();
```

```
    for i in 0..n {
        roots.insert(d.find(i));
    }
    roots.len()
}

#[cfg(test)]
mod tests_num_comp {
    use super::num_components_dsu;

    #[test]
    fn counts() {
        let edges = [(0, 1), (2, 3)];
        assert_eq!(num_components_dsu(5, &edges), 3);
    }
}
```

Pattern C: shortest path in unweighted graphs

Use BFS (Chapter 13).

Pattern D: shortest path in weighted graphs

Use Dijkstra if weights are non-negative. If weights can be negative, Dijkstra is invalid.

Pattern E: incremental connectivity as edges arrive

DSU supports incremental edge additions naturally:

- add edges one-by-one with `union`,
- query `same` at any time.

Example: earliest time two nodes become connected

Suppose edges arrive with timestamps, find earliest timestamp when two nodes connect.

```rust
#[derive(Clone, Copy, Debug)]
pub struct TimedEdge {
    pub t: i64,
    pub u: usize,
    pub v: usize,
}

pub fn earliest_connection(n: usize, mut edges: Vec<TimedEdge>, a: usize, b: usize) ->
↪  Option<i64> {
    edges.sort_unstable_by(|x, y| x.t.cmp(&y.t));

    let mut d = super::DSU::new(n);
    for e in edges {
        d.union(e.u, e.v);
        if d.same(a, b) {
            return Some(e.t);
        }
    }
    None
}

#[cfg(test)]
mod tests_earliest {
    use super::{earliest_connection, TimedEdge};

    #[test]
    fn finds_time() {
        let edges = vec![
            TimedEdge { t: 5, u: 0, v: 1 },
            TimedEdge { t: 7, u: 1, v: 2 },
```

```
            TimedEdge { t: 9, u: 3, v: 4 },
            TimedEdge { t: 10, u: 2, v: 3 },
        ];
        assert_eq!(earliest_connection(5, edges, 0, 4), Some(10));
    }
}
```

14.4 Complexity Analysis

Dijkstra

Let V be number of vertices, E edges. With adjacency lists and a binary heap:

- Each edge can relax at most once to improve a distance.
- Each improvement pushes a new heap entry.
- Heap operations cost $O(\log V)$.

Practical bound:

$$O((V + E) \log V)$$

Memory:

- $O(V)$ for distance arrays and parents,
- $O(E)$ for adjacency lists,
- heap holds up to $O(E)$ entries in the stale-entry implementation.

Union-Find

With path compression + union by size:

- amortized time per operation is very close to constant,
- theoretical bound is $O(\alpha(n))$ where α is the inverse Ackermann function.

For engineering purposes:

- treat DSU operations as effectively $O(1)$ amortized.

Connectivity queries

If you have Q connectivity queries:

- BFS per query: $O(Q \cdot (V + E))$
- DSU build once: $O(E \cdot \alpha(V))$ then queries $O(Q \cdot \alpha(V))$

So DSU dominates when Q is large.

14.5 Chapter Conclusion

This chapter connected shortest-path and connectivity engineering in Rust:

- You implemented Dijkstra using `BinaryHeap` + `Reverse` and the stale-entry technique, with optional parent pointers for path reconstruction.
- You implemented a production-grade DSU (Union-Find) using path compression and union-by-size.
- You mapped common connectivity problems to the correct tool: BFS/DFS, Dijkstra, or DSU, and implemented real query patterns including earliest connection time.

- You performed precise complexity reasoning: Dijkstra is $O((V + E) \log V)$ with a heap, and DSU operations are effectively constant amortized time.

With these foundations, the next step is advanced graph engineering: minimum spanning trees, topological sorting, and strongly connected components, all built on the same disciplined data structures.

String Pattern Matching

15.1 Naive Pattern Matching

Pattern matching answers the fundamental question:

> Where does a pattern `p` occur inside a text `t`?

Problem definition

Given:

- text T of length n,
- pattern P of length m,

find all indices i such that:

$$T[i..i+m) = P[0..m)$$

(when $i + m \leq n$).

Naive idea

Try every alignment of the pattern against the text:

- for each start position i in the text,

- compare characters until mismatch or full match.

Time complexity:

$$O(n \cdot m)$$

Worst-case occurs when the text contains many repeated prefixes of the pattern.

Engineering note: bytes vs Unicode

Rust `&str` is UTF-8 and cannot be indexed by integer. For algorithmic pattern matching:

- use bytes (`as_bytes()`) for ASCII/binary data and exact byte matching,
- use `chars()` only if you truly want Unicode scalar matching (slower and not constant-width).

Most systems pattern matching (protocols, tokens, logs) is byte-based.

Naive match on bytes: find all occurrences

```rust
pub fn naive_find_all(text: &[u8], pat: &[u8]) -> Vec<usize> {
    let n = text.len();
    let m = pat.len();
    if m == 0 { return (0..=n).collect(); }
    if m > n { return Vec::new(); }

    let mut out = Vec::new();

    for i in 0..=(n - m) {
        let mut ok = true;
        for j in 0..m {
            if text[i + j] != pat[j] {
                ok = false;
```

```
                break;
            }
        }
        if ok {
            out.push(i);
        }
    }

    out
}

#[cfg(test)]
mod tests_naive {
    use super::naive_find_all;

    #[test]
    fn finds() {
        let t = b"ababa";
        let p = b"aba";
        assert_eq!(naive_find_all(t, p), vec![0, 2]);
    }

    #[test]
    fn empty_pattern() {
        let t = b"abc";
        let p = b"";
        assert_eq!(naive_find_all(t, p), vec![0, 1, 2, 3]);
    }

    #[test]
    fn no_match() {
        assert_eq!(naive_find_all(b"aaaa", b"b"), Vec::<usize>::new());
    }
```

```
}
```

When naive is good enough

Naive matching can be the right engineering choice when:

- pattern is short,
- text is short or few queries,
- readability matters more than worst-case,
- you can early-break quickly in typical data.

When worst-case matters or you match repeatedly, use KMP.

15.2 KMP Algorithm

Knuth–Morris–Pratt (KMP) eliminates redundant comparisons by reusing information about the pattern. It guarantees:

$$O(n + m)$$

in the worst case.

Idea

When a mismatch happens at pattern position j, KMP does not restart at $j = 0$. Instead, it jumps to the longest proper prefix of the pattern that is also a suffix of the already matched prefix. This jump is derived from the **prefix function** (also known as LPS array).

KMP outputs all matches

KMP can find all occurrences including overlaps.

Implementation plan

- Precompute `pi` (prefix function) for the pattern.
- Scan the text with a running match length `j`.
- On mismatch, reduce `j` using `pi`.
- On full match, report index and continue with `pi[m-1]` for overlaps.

KMP: find all occurrences (bytes)

```
pub fn kmp_find_all(text: &[u8], pat: &[u8]) -> Vec<usize> {
    let n = text.len();
    let m = pat.len();

    if m == 0 { return (0..=n).collect(); }
    if m > n { return Vec::new(); }

    let pi = prefix_function(pat);
    let mut out = Vec::new();

    let mut j = 0usize; /* current matched length in pattern */

    for i in 0..n {
        while j > 0 && text[i] != pat[j] {
            j = pi[j - 1];
        }
        if text[i] == pat[j] {
```

```
            j += 1;
        }
        if j == m {
            out.push(i + 1 - m);
            j = pi[m - 1]; /* allow overlaps */
        }
    }

    out
}

#[cfg(test)]
mod tests_kmp {
    use super::kmp_find_all;

    #[test]
    fn overlaps() {
        assert_eq!(kmp_find_all(b"aaaaa", b"aa"), vec![0, 1, 2, 3]);
    }

    #[test]
    fn basic() {
        assert_eq!(kmp_find_all(b"ababa", b"aba"), vec![0, 2]);
    }

    #[test]
    fn no_match() {
        assert_eq!(kmp_find_all(b"abc", b"d"), Vec::<usize>::new());
    }
}
```

15.3 Understanding the Prefix Function

The prefix function for pattern P (length m) is an array `pi[0..m)` where:

$$\pi[i] = \text{length of the longest proper prefix of } P[0..i] \text{ that is also a suffix of } P[0..i]$$

Proper prefix means it cannot equal the entire string.

What it encodes

If we matched j characters and fail at j, then:

- the next best candidate match length is `pi[j-1]`,
- because the first `pi[j-1]` chars equal the last `pi[j-1]` chars of what we matched.

Prefix function computation (O(m))

```rust
pub fn prefix_function(pat: &[u8]) -> Vec<usize> {
    let m = pat.len();
    let mut pi = vec![0usize; m];

    for i in 1..m {
        let mut j = pi[i - 1];

        while j > 0 && pat[i] != pat[j] {
            j = pi[j - 1];
        }

        if pat[i] == pat[j] {
            j += 1;
        }
```

```
        pi[i] = j;
    }

    pi
}

#[cfg(test)]
mod tests_prefix {
    use super::prefix_function;

    #[test]
    fn known_example() {
        /* pattern: a b a b a c */
        let p = b"ababac";
        let pi = prefix_function(p);
        assert_eq!(pi, vec![0, 0, 1, 2, 3, 0]);
    }

    #[test]
    fn all_same() {
        let p = b"aaaaa";
        let pi = prefix_function(p);
        assert_eq!(pi, vec![0, 1, 2, 3, 4]);
    }
}
```

Concrete mental model

Maintain a variable `j` = length of current best prefix match for the prefix ending at `i-1`. When you extend to `i`:

- try to extend match: compare `pat[i]` with `pat[j]`.

- if mismatch, shrink `j` to `pi[j-1]` and try again.
- if match, increment `j`.

This is the same logic used during KMP text scanning. KMP is essentially reusing this internal automaton.

15.4 Practical Use Cases

Pattern matching is not only an academic exercise. It appears in real engineering work:

Use case 1: scanning logs and telemetry

You can scan a large stream for a fixed marker (e.g., error signatures). KMP is useful when:

- the text is huge,
- the pattern is fixed,
- worst-case behavior matters (repetitive text).

Use case 2: protocol parsing

Binary protocols often contain fixed byte sequences:

- framing markers,
- magic headers,
- delimiters.

Byte-based KMP is a clean fit.

Use case 3: repeated queries against the same pattern

If you apply the same pattern across many texts:

- precompute `pi` once,
- scan each text in $O(n)$.

Use case 4: detect periodicity in a string

A classic trick uses the prefix function: If m is pattern length and $k = m - \pi[m-1]$, then the string is periodic with period k when $m \bmod k = 0$.

```rust
pub fn smallest_period_len(s: &[u8]) -> usize {
    if s.is_empty() { return 0; }
    let pi = prefix_function(s);
    let m = s.len();
    let k = m - pi[m - 1];
    if k != 0 && m % k == 0 { k } else { m }
}

#[cfg(test)]
mod tests_period {
    use super::smallest_period_len;

    #[test]
    fn periodic() {
        assert_eq!(smallest_period_len(b"ababab"), 2);
        assert_eq!(smallest_period_len(b"aaaa"), 1);
        assert_eq!(smallest_period_len(b"abcab"), 5);
    }
}
```

Use case 5: building a simple substring search API

This function accepts `&str` but matches on bytes, returning byte indices. For ASCII log data this is often exactly what you want.

```rust
pub fn find_all_str(text: &str, pat: &str) -> Vec<usize> {
    kmp_find_all(text.as_bytes(), pat.as_bytes())
}

#[cfg(test)]
mod tests_find_all_str {
    use super::find_all_str;

    #[test]
    fn works() {
        assert_eq!(find_all_str("ababa", "aba"), vec![0, 2]);
    }
}
```

Engineering warning

Returning byte indices is correct for byte-based algorithms. If you require character indices for general Unicode, you must convert carefully and accept extra cost.

15.5 Chapter Conclusion

This chapter established string pattern matching as a disciplined engineering tool:

- Naive matching is simple and often sufficient, but has $O(nm)$ worst-case behavior.
- KMP guarantees $O(n + m)$ by avoiding redundant comparisons and enabling overlap matches.

- The prefix function (`pi`) encodes the fallback transitions of the pattern automaton.
- Practical use cases include log scanning, protocol parsing, repeated query matching, and periodicity detection.

With KMP mastered, you can move to higher-level string algorithms (Z-function, rolling hash, suffix arrays/automata) when your domain demands faster multi-pattern or advanced substring analytics.

Capstone Mini Project

16.1 Designing a Small Query Engine

This capstone builds a small in-memory query engine that combines multiple algorithmic techniques from this handbook. The goal is not to compete with databases; the goal is to practice **engineering discipline**:

- define data layout intentionally,
- build reusable indices,
- answer queries efficiently,
- measure performance,
- refactor into clean architecture.

Project scope

We will implement an in-memory dataset of `Record`s and support queries like:

- filter by exact match: `country == "SA"`,
- filter by range: `age in [30, 45]`,

- substring match: `name contains "man"`,
- combine filters: `country == "SA" AND age in [30,45] AND name contains "man"`.

Design constraints

- Pure Rust stable, Windows-friendly, no OS-specific syscalls required.
- Algorithms are implemented using std only: `HashMap`, `BTreeMap`, binary search, prefix sums, and KMP.
- Emphasis on clarity + performance reasoning.

Core record model

We will store owned strings (`String`) for long-lived data. We also store an integer id for stable indexing.

```rust
#[derive(Clone, Debug)]
pub struct Record {
    pub id: u32,
    pub name: String,
    pub country: String,
    pub age: u32,
    pub score: i64,
}

impl Record {
    pub fn new(id: u32, name: &str, country: &str, age: u32, score: i64) -> Self {
        Self {
            id,
            name: name.to_string(),
            country: country.to_string(),
```

```
            age,
            score,
        }
    }
}
```

Data layout choices

We choose a **column-friendly** structure while still keeping `Records`:

- store `Vec<Record>` as the source of truth,
- store indices into the vector for query results,
- build indices mapping field values to record indices.

This reduces data copying and makes filters cheap.

Query language (minimal)

We define a small query object that represents a conjunction of optional filters.

```rust
#[derive(Clone, Debug, Default)]
pub struct Query {
    pub country_eq: Option<String>,
    pub age_range: Option<(u32, u32)>,     /* inclusive */
    pub name_contains: Option<String>,     /* substring */
    pub min_score: Option<i64>,            /* score >= min_score */
}
```

Engine structure

The engine stores:

- records,
- a hashing index for exact match,
- an ordered index for range queries,
- optional precomputed helpers.

```
use std::collections::{HashMap, BTreeMap};

pub struct Engine {
    records: Vec<Record>,

    /* exact match: country -> list of record indices */
    idx_country: HashMap<String, Vec<usize>>,

    /* range query: age -> list of record indices; BTreeMap gives range() */
    idx_age: BTreeMap<u32, Vec<usize>>,
}

impl Engine {
    pub fn new(records: Vec<Record>) -> Self {
        let mut e = Self {
            records,
            idx_country: HashMap::new(),
            idx_age: BTreeMap::new(),
        };
        e.build_indices();
        e
    }
```

```
    fn build_indices(&mut self) {
        self.idx_country.clear();
        self.idx_age.clear();

        for (i, r) in self.records.iter().enumerate() {
            self.idx_country.entry(r.country.clone()).or_insert_with(Vec::new).push(i);
            self.idx_age.entry(r.age).or_insert_with(Vec::new).push(i);
        }
    }

    pub fn get(&self, idx: usize) -> &Record {
        &self.records[idx]
    }

    pub fn len(&self) -> usize {
        self.records.len()
    }
}
```

Correctness first

We will implement:

- a naive scan engine (reference),
- an indexed engine (optimized),

and test that both return the same results.

16.2 Combining Hashing, Range Queries, and Search

A query engine is a pipeline:

1. choose an initial candidate set using the cheapest / most selective index,
2. intersect with other indexed constraints,
3. apply non-indexed filters (like substring search),
4. return results.

Set representation

We represent candidate sets as a boolean bitmap for fast intersection:

- `Vec<bool>` is simple but has special bit storage behavior,
- `Vec<u8>` is predictable and often faster for tight loops.

We use `Vec<u8>` where 0/1 indicates membership.

Helper: mark indices into a bitmap

```
pub fn mark_bitmap(n: usize, indices: &[usize]) -> Vec<u8> {
    let mut b = vec![0u8; n];
    for &i in indices {
        b[i] = 1;
    }
    b
}

pub fn bitmap_and_inplace(a: &mut [u8], b: &[u8]) {
    assert_eq!(a.len(), b.len());
    for i in 0..a.len() {
        a[i] &= b[i];
    }
}
```

Range query on age using BTreeMap

We gather indices for all ages in `[lo, hi]`.

```
impl Engine {
    fn indices_age_range(&self, lo: u32, hi: u32) -> Vec<usize> {
        let mut out: Vec<usize> = Vec::new();
        for (_age, idxs) in self.idx_age.range(lo..=hi) {
            out.extend_from_slice(idxs);
        }
        out
    }
}
```

Substring search: KMP on bytes

We reuse KMP from Chapter 15 to implement `name contains`. For simplicity, we do a byte-based substring match.

```
pub fn prefix_function(pat: &[u8]) -> Vec<usize> {
    let m = pat.len();
    let mut pi = vec![0usize; m];
    for i in 1..m {
        let mut j = pi[i - 1];
        while j > 0 && pat[i] != pat[j] {
            j = pi[j - 1];
        }
        if pat[i] == pat[j] {
            j += 1;
        }
        pi[i] = j;
    }
    pi
```

```
}

pub fn kmp_contains(text: &[u8], pat: &[u8]) -> bool {
    let n = text.len();
    let m = pat.len();
    if m == 0 { return true; }
    if m > n { return false; }

    let pi = prefix_function(pat);
    let mut j = 0usize;

    for i in 0..n {
        while j > 0 && text[i] != pat[j] {
            j = pi[j - 1];
        }
        if text[i] == pat[j] {
            j += 1;
        }
        if j == m {
            return true;
        }
    }
    false
}
```

Naive reference query (full scan)

This is the correctness oracle.

```
impl Engine {
    pub fn query_naive(&self, q: &Query) -> Vec<usize> {
        let mut out = Vec::new();
```

```
        for i in 0..self.records.len() {
            let r = &self.records[i];

            if let Some(ref c) = q.country_eq {
                if &r.country != c { continue; }
            }

            if let Some((lo, hi)) = q.age_range {
                if r.age < lo || r.age > hi { continue; }
            }

            if let Some(mins) = q.min_score {
                if r.score < mins { continue; }
            }

            if let Some(ref needle) = q.name_contains {
                if !kmp_contains(r.name.as_bytes(), needle.as_bytes()) { continue; }
            }

            out.push(i);
        }

        out
    }
}
```

Optimized indexed query

We build a candidate bitmap from available indices, then verify remaining predicates. Strategy:

- if `country_eq` exists, start from that candidate set,

- else if `age_range` exists, start from it,
- else start from all records.

Then apply remaining filters.

```
impl Engine {
    pub fn query_indexed(&self, q: &Query) -> Vec<usize> {
        let n = self.records.len();

        /* 1) choose initial candidate set */
        let mut cand: Vec<u8> = vec![1u8; n];

        let mut seeded = false;

        if let Some(ref c) = q.country_eq {
            if let Some(idxs) = self.idx_country.get(c) {
                cand = mark_bitmap(n, idxs);
            } else {
                return Vec::new();
            }
            seeded = true;
        }

        if let Some((lo, hi)) = q.age_range {
            let idxs = self.indices_age_range(lo, hi);
            let b = mark_bitmap(n, &idxs);
            if seeded {
                bitmap_and_inplace(&mut cand, &b);
            } else {
                cand = b;
                seeded = true;
            }
        }
```

```
        /* 2) apply non-indexed filters by scanning candidates */
        let mut out: Vec<usize> = Vec::new();

        for i in 0..n {
            if cand[i] == 0 { continue; }
            let r = &self.records[i];

            if let Some(mins) = q.min_score {
                if r.score < mins { continue; }
            }

            if let Some(ref needle) = q.name_contains {
                if !kmp_contains(r.name.as_bytes(), needle.as_bytes()) { continue; }
            }

            out.push(i);
        }

        out
    }
}
```

Correctness testing: naive vs indexed

```
#[cfg(test)]
mod tests_engine_correctness {
    use super::{Engine, Query, Record};

    fn sample_engine() -> Engine {
        let rows = vec![
            Record::new(1, "Ayman", "SA", 41, 95),
```

```
            Record::new(2, "Hassan", "SA", 33, 70),
            Record::new(3, "Mariam", "EG", 29, 88),
            Record::new(4, "Salman", "SA", 27, 60),
            Record::new(5, "Omar", "EG", 44, 99),
            Record::new(6, "Iman", "SA", 45, 85),
        ];
        Engine::new(rows)
    }

    #[test]
    fn naive_matches_indexed() {
        let e = sample_engine();

        let mut q = Query::default();
        q.country_eq = Some("SA".to_string());
        q.age_range = Some((30, 45));
        q.name_contains = Some("man".to_string());
        q.min_score = Some(50);

        let a = e.query_naive(&q);
        let b = e.query_indexed(&q);

        assert_eq!(a, b);
    }

    #[test]
    fn various_queries() {
        let e = sample_engine();

        let q1 = Query { country_eq: Some("EG".into()), ..Default::default() };
        assert_eq!(e.query_naive(&q1), e.query_indexed(&q1));

        let q2 = Query { age_range: Some((40, 50)), ..Default::default() };
```

```
        assert_eq!(e.query_naive(&q2), e.query_indexed(&q2));

        let q3 = Query { name_contains: Some("a".into()), ..Default::default() };
        assert_eq!(e.query_naive(&q3), e.query_indexed(&q3));

        let q4 = Query { min_score: Some(90), ..Default::default() };
        assert_eq!(e.query_naive(&q4), e.query_indexed(&q4));
    }
}
```

16.3 Performance Evaluation

Performance evaluation is an engineering workflow:

- define what you measure (latency, throughput),
- isolate variables,
- warm up,
- measure repeatedly,
- interpret results with complexity and memory behavior in mind.

Practical timing with `std::time::Instant`

This is not a full benchmark harness, but it is sufficient to compare naive vs indexed in a controlled run.

```
use std::time::Instant;

pub fn time_queries(e: &Engine, q: &Query, iters: usize) -> (u128, u128) {
```

```
    let mut sink: usize = 0;

    let t0 = Instant::now();
    for _ in 0..iters {
        let r = e.query_naive(q);
        sink ^= r.len();
    }
    let naive_ns = t0.elapsed().as_nanos();

    let t1 = Instant::now();
    for _ in 0..iters {
        let r = e.query_indexed(q);
        sink ^= r.len();
    }
    let indexed_ns = t1.elapsed().as_nanos();

    /* prevent sink from being optimized away */
    if sink == usize::MAX { eprintln!("{}", sink); }

    (naive_ns, indexed_ns)
}

#[cfg(test)]
mod tests_timing_smoke {
    use super::{time_queries, Engine, Query, Record};

    #[test]
    fn timing_runs() {
        let rows = (0..10_000u32).map(|i| {
            let country = if i % 3 == 0 { "SA" } else if i % 3 == 1 { "EG" } else { "AE" };
            let name = if i % 10 == 0 { "Salman" } else { "User" };
            Record::new(i, name, country, (20 + (i % 40)) as u32, (i as i64) % 100)
        }).collect();
```

```
        let e = Engine::new(rows);

        let q = Query {
            country_eq: Some("SA".into()),
            age_range: Some((30, 45)),
            name_contains: Some("man".into()),
            min_score: Some(10),
        };

        let (naive_ns, indexed_ns) = time_queries(&e, &q, 50);
        assert!(naive_ns > 0);
        assert!(indexed_ns > 0);
    }
}
```

Interpreting results

Expect the indexed version to win when:

- the filter is selective (small candidate set),
- the dataset is large,
- substring checks are expensive and can be avoided for most records.

Expect naive to compete when:

- the dataset is small,
- most filters are absent,
- most records match anyway (no selectivity).

Performance knobs

- **Index selectivity**: choose best index first.
- **Bitmap vs Vec of indices**: for tiny candidate sets, a Vec of indices can be faster than scanning a bitmap.
- **Memory layout**: CSR-like adjacency ideas apply: fewer allocations, more contiguous data.
- **Substring matching**: KMP is worst-case linear, but constant factors matter.

16.4 Refactoring for Clean Architecture

Algorithms are only half the project. The other half is clean architecture: readable, testable, and maintainable code.

Refactoring goals

- separate indexing from query evaluation,
- isolate data model from algorithms,
- keep pure functions where possible,
- introduce traits to enable alternative index implementations.

Suggested module layout

In a real crate you would split into:

- `model.rs`: `Record`, `Query`

- `index.rs`: `CountryIndex`, `AgeIndex`
- `search.rs`: substring algorithms (KMP)
- `engine.rs`: orchestrates indices + filtering
- `bench.rs`: timing harness

Below we illustrate this separation with traits while keeping everything in one file.

Trait-based index interface

```
pub trait CandidateIndex {
    fn candidates(&self, q: &Query, n: usize) -> Option<Vec<u8>>;
}

/* Country exact-match index */
pub struct CountryIndex {
    m: std::collections::HashMap<String, Vec<usize>>,
}

impl CountryIndex {
    pub fn build(records: &[Record]) -> Self {
        let mut m = std::collections::HashMap::new();
        for (i, r) in records.iter().enumerate() {
            m.entry(r.country.clone()).or_insert_with(Vec::new).push(i);
        }
        Self { m }
    }
}

impl CandidateIndex for CountryIndex {
    fn candidates(&self, q: &Query, n: usize) -> Option<Vec<u8>> {
        let c = q.country_eq.as_ref()?;
```

```
        let idxs = self.m.get(c)?;
        Some(mark_bitmap(n, idxs))
    }
}

/* Age range index */
pub struct AgeIndex {
    m: std::collections::BTreeMap<u32, Vec<usize>>,
}

impl AgeIndex {
    pub fn build(records: &[Record]) -> Self {
        let mut m = std::collections::BTreeMap::new();
        for (i, r) in records.iter().enumerate() {
            m.entry(r.age).or_insert_with(Vec::new).push(i);
        }
        Self { m }
    }

    fn range_indices(&self, lo: u32, hi: u32) -> Vec<usize> {
        let mut out = Vec::new();
        for (_age, idxs) in self.m.range(lo..=hi) {
            out.extend_from_slice(idxs);
        }
        out
    }
}

impl CandidateIndex for AgeIndex {
    fn candidates(&self, q: &Query, n: usize) -> Option<Vec<u8>> {
        let (lo, hi) = q.age_range?;
        let idxs = self.range_indices(lo, hi);
        Some(mark_bitmap(n, &idxs))
```

```
    }
}
```

Engine refactor: combine indices

We can now evaluate:

- get candidate bitmaps from each index if the query uses that field,
- intersect them,
- scan remaining predicates.

```
pub struct Engine2 {
    records: Vec<Record>,
    idx_country: CountryIndex,
    idx_age: AgeIndex,
}

impl Engine2 {
    pub fn new(records: Vec<Record>) -> Self {
        let idx_country = CountryIndex::build(&records);
        let idx_age = AgeIndex::build(&records);
        Self { records, idx_country, idx_age }
    }

    pub fn query(&self, q: &Query) -> Vec<usize> {
        let n = self.records.len();
        let mut cand: Option<Vec<u8>> = None;

        for idx in [&self.idx_country as &dyn CandidateIndex, &self.idx_age as &dyn
        ↪  CandidateIndex] {
```

```
            if let Some(b) = idx.candidates(q, n) {
                cand = match cand {
                    None => Some(b),
                    Some(mut cur) => { bitmap_and_inplace(&mut cur, &b); Some(cur) }
                };
            }
        }

        let cand = cand.unwrap_or_else(|| vec![1u8; n]);

        let mut out = Vec::new();
        for i in 0..n {
            if cand[i] == 0 { continue; }
            let r = &self.records[i];

            if let Some(mins) = q.min_score {
                if r.score < mins { continue; }
            }
            if let Some(ref needle) = q.name_contains {
                if !kmp_contains(r.name.as_bytes(), needle.as_bytes()) { continue; }
            }
            out.push(i);
        }
        out
    }

    pub fn query_naive(&self, q: &Query) -> Vec<usize> {
        let mut out = Vec::new();
        for i in 0..self.records.len() {
            let r = &self.records[i];
            if let Some(ref c) = q.country_eq { if &r.country != c { continue; } }
            if let Some((lo, hi)) = q.age_range { if r.age < lo || r.age > hi { continue; } }
            if let Some(mins) = q.min_score { if r.score < mins { continue; } }
```

```
            if let Some(ref needle) = q.name_contains {
                if !kmp_contains(r.name.as_bytes(), needle.as_bytes()) { continue; }
            }
            out.push(i);
        }
        out
    }
}

#[cfg(test)]
mod tests_engine2 {
    use super::{Engine2, Query, Record};

    #[test]
    fn matches_naive() {
        let rows = vec![
            Record::new(1, "Ayman", "SA", 41, 95),
            Record::new(2, "Hassan", "SA", 33, 70),
            Record::new(3, "Mariam", "EG", 29, 88),
            Record::new(4, "Salman", "SA", 27, 60),
            Record::new(5, "Omar", "EG", 44, 99),
            Record::new(6, "Iman", "SA", 45, 85),
        ];
        let e = Engine2::new(rows);

        let q = Query {
            country_eq: Some("SA".into()),
            age_range: Some((30, 45)),
            name_contains: Some("man".into()),
            min_score: Some(10),
        };

        assert_eq!(e.query_naive(&q), e.query(&q));
```

```
    }
}
```

Clean architecture takeaways

- Indices become interchangeable components.
- Query evaluation becomes a simple orchestration layer.
- Algorithms (KMP, bitmap intersections) remain pure and testable.

16.5 Final Project Conclusion

This capstone demonstrated how algorithm knowledge becomes engineering capability:

- You designed a small query engine with an explicit data model and constraints.
- You combined hashing (`HashMap`) for exact match, ordered range queries (`BTreeMap`), and substring search (KMP).
- You built a correctness oracle (naive scan) and verified the indexed engine against it with unit tests.
- You performed realistic performance evaluation using `Instant`, learned how selectivity drives speed, and identified practical optimization knobs.
- You refactored into clean architecture using traits and isolated index responsibilities for maintainable growth.

The most important result is not the code itself, but the mindset:

Build indices intentionally, minimize candidate sets early, measure performance, and refactor into clean components.

This is the core discipline behind practical algorithm engineering in Rust.

Appendices

Appendix A: Complexity Cheat Sheet

This appendix is a practical complexity reference for the patterns used in this handbook. Let n be input size (items, characters, vertices), m be pattern length, V vertices, E edges, C capacity, and Q queries.

Core growth rates (intuition)

- $O(1)$: constant work
- $O(\log n)$: balanced tree / heap operation
- $O(n)$: single pass
- $O(n \log n)$: sorting + divide-and-conquer patterns
- $O(n^2)$: double loop over pairs
- $O(2^n)$: subset enumeration (avoid for large n)

Common data structure operations

- `Vec`:

  - indexing: $O(1)$
  - push (amortized): $O(1)$
  - pop: $O(1)$
  - insert/remove in middle: $O(n)$
  - binary search on sorted Vec: $O(\log n)$
- `HashMap` (average-case):
  - insert/lookup/remove: $O(1)$ average, $O(n)$ worst-case
- `BTreeMap`:
  - insert/lookup/remove: $O(\log n)$
  - range queries: $O(\log n + k)$ where k results returned
- `BinaryHeap`:
  - push/pop: $O(\log n)$
  - peek: $O(1)$
  - build from Vec (heapify): $O(n)$
- Sorting:
  - `sort_unstable`: $O(n \log n)$ average, in-place, not stable
  - `sort`: $O(n \log n)$, stable

Algorithm patterns in this handbook

- Two pointers / sliding window: $O(n)$
- Prefix sum:

  - build: $O(n)$
  - range query: $O(1)$ per query
- Binary search:
  - search in sorted array: $O(\log n)$
  - binary search on answer: $O(\log \text{answer\_range}) \times$ check cost
- Monotonic stack (NGE, histogram): $O(n)$ (each index pushed/popped once)
- KMP string search: $O(n + m)$
- BFS/DFS: $O(V + E)$
- Dijkstra with binary heap: $O((V + E) \log V)$
- DSU (Union-Find): near $O(1)$ amortized per operation
- Knapsack:
  - 0/1: $O(n \cdot C)$ time, $O(C)$ space (optimized)
  - unbounded: $O(n \cdot C)$ time, $O(C)$ space

Engineering checklist

- If your solution is $O(n^2)$, validate n limits and typical data size.
- Watch hidden constants: hashing, allocations, cache misses, string conversions.
- Prefer single allocation + linear passes over many small allocations.

Appendix B: Common Rust Pitfalls in Algorithm Design

This appendix lists the mistakes that most often break correctness or performance in Rust algorithm code.

Pitfall 1: indexing `&str` by integer

Rust UTF-8 strings are not random-access by character index. Use:

- bytes: `text.as_bytes()` for byte algorithms and protocol/log parsing,
- `chars()` if you truly need Unicode scalar iteration,
- explicit conversion to `Vec<char>` only when necessary and acceptable.

```rust
pub fn count_ascii_a(s: &str) -> usize {
    s.as_bytes().iter().filter(|&&b| b == b'a' || b == b'A').count()
}
```

Pitfall 2: accidental quadratic time from `String` concatenation

Repeated `push_str` or `format!` in loops can reallocate many times. Use `String::with_capacity` or `Vec<u8>` buffer.

```rust
pub fn join_words(words: &[&str]) -> String {
    let total: usize = words.iter().map(|w| w.len()).sum();
    let mut s = String::with_capacity(total);
    for w in words {
        s.push_str(w);
    }
    s
}
```

Pitfall 3: cloning large vectors unnecessarily

`clone()` copies data. Prefer:

- pass slices: `&[T]`,
- reuse buffers, pass mutable references,
- move ownership once when needed.

Pitfall 4: wrong heap direction

`BinaryHeap` is a max-heap. For min-heap behavior, use `Reverse` or implement custom `Ord`.

```
use std::cmp::Reverse;
use std::collections::BinaryHeap;

pub fn pop_in_ascending(v: &[i32]) -> Vec<i32> {
    let mut h: BinaryHeap<Reverse<i32>> = BinaryHeap::new();
    for &x in v { h.push(Reverse(x)); }
    let mut out = Vec::new();
    while let Some(Reverse(x)) = h.pop() { out.push(x); }
    out
}
```

Pitfall 5: integer overflow in mid-point or distance sums

Use safe midpoint:

$$mid = lo + (hi - lo)/2$$

and use `i64` or `u64` for sums and distances.

```
pub fn mid(lo: usize, hi: usize) -> usize {
    lo + (hi - lo) / 2
}
```

Pitfall 6: recursion depth in DFS and DP

Rust does not guarantee deep recursion safety. Prefer iterative DFS and bottom-up DP for large inputs.

Pitfall 7: choosing the wrong comparator for duplicates

Monotonic stack and boundary problems depend on strict vs non-strict comparisons. Always test:

- strictly increasing,
- strictly decreasing,
- all equal,
- mixed duplicates.

Pitfall 8: `HashMap` key ownership and lifetime confusion

Avoid storing references with complex lifetimes unless you must. Most algorithmic indices are easiest as owned keys (`String`, `u32`, etc.).

Appendix C: Memory and Allocation Best Practices

Allocation and memory layout often dominate algorithm performance on real workloads.

Rule 1: reserve capacity

If you can estimate size, allocate once.

```rust
pub fn build_vec(n: usize) -> Vec<i32> {
    let mut v = Vec::with_capacity(n);
    for i in 0..n {
        v.push(i as i32);
    }
    v
}
```

Rule 2: prefer contiguous storage

`Vec<T>` is cache-friendly. For graphs, consider CSR-style layouts if the graph is huge.

Rule 3: reuse buffers

Allocate buffers once and reuse across iterations (especially in parsing/search pipelines).

```rust
pub fn reuse_buffer(chunks: &[&[u8]]) -> usize {
    let mut buf: Vec<u8> = Vec::with_capacity(1024);
    let mut total = 0usize;

    for c in chunks {
        buf.clear();
        buf.extend_from_slice(c);
        total += buf.len();
    }

    total
}
```

Rule 4: choose data structures by access pattern

- many lookups by key: `HashMap`

- ordered keys + range: `BTreeMap`
- top-k: `BinaryHeap`
- frequent push/pop ends: `VecDeque`

Rule 5: avoid per-element heap allocations

Avoid `Vec<Vec<T>` for huge graphs if it creates too many small allocations. Consider:

- CSR representation,
- storing edges in one flat `Vec` with offsets.

Rule 6: avoid converting between `String` and `Vec<char>`

Conversions allocate and often destroy cache locality. Prefer byte-based algorithms when semantics allow.

Rule 7: prefer small, copyable keys

If heap or map keys are large structs, store:

- indices,
- lightweight keys,
- or `Box<T>` when necessary.

Rule 8: reduce memory for DP

- rolling variables for k-dependent recurrences,
- 2D to 1D compression when dependencies are row-local,

- store only needed reconstruction info.

Appendix D: Categorized Practice Problems

This appendix is a pattern-based practice plan. Each category lists problems to solve with increasing difficulty. You can map these to any platform or your own custom datasets. The goal is to practice **the pattern**, not a specific website.

Two pointers / sliding window

- Two-sum on sorted array
- Remove duplicates in-place
- Longest substring without repeating characters
- Minimum window substring (advanced)

Prefix sums / range techniques

- Subarray sum equals K
- 2D prefix sum for matrix sum queries
- Difference array for range updates
- Count of subarrays with sum in range (advanced)

Binary search

- Lower/upper bound practice

- Search in rotated sorted array
- Binary search on answer (minimum feasible value)
- K-th smallest in sorted matrix (advanced)

Stacks / monotonic stacks

- Next greater element
- Daily temperatures
- Largest rectangle in histogram
- Maximal rectangle in binary matrix (advanced)

Heaps / greedy

- Top-K elements (streaming)
- Merge K sorted lists / arrays
- Course scheduling by deadlines
- Minimum number of meeting rooms (advanced)

Dynamic programming

- Fibonacci / stairs
- House robber variants
- 0/1 knapsack

- Unbounded knapsack / coin change
- Longest increasing subsequence (advanced)

Graphs

- BFS shortest path in unweighted graph
- Connected components
- Cycle detection (directed and undirected)
- Dijkstra shortest path (non-negative weights)
- Minimum spanning tree with DSU (advanced)

Strings

- Naive substring search
- KMP prefix function
- Periodicity detection via prefix function
- Z-function (advanced)

Practice discipline

For every solved problem:

- write the state/transition (DP) or invariant (stack/graph),
- write 5 tests: empty, small, duplicates, worst-case shape, random,
- measure complexity and note memory allocations.

Appendix E: Algorithm Pattern Summary Table

Pattern summary

Pattern	When to use	Typical complexity
Two pointers	sorted arrays, in-place removal, partitioning	$O(n)$ time, $O(1)$ extra
Sliding window	fixed/variable window with condition, longest/shortest subarray	$O(n)$ amortized
Prefix sum	fast range sum queries, subarray sum counting	build $O(n)$, query $O(1)$
Difference array	many range updates, apply later	update $O(1)$, finalize $O(n)$
Binary search	monotonic predicate, sorted arrays, answer search	$O(\log n)$ or $O(\log R) \cdot check$
Hashing	frequency counting, grouping, membership tests	avg $O(1)$ per op
Monotonic stack	nearest greater/smaller, range boundaries, histogram	$O(n)$
Heap / priority queue	top-k, scheduling, best-next choice	$O(\log n)$ per push/pop
DP (1D)	optimal substructure over prefix or time	often $O(n)$ or $O(n \cdot C)$
Knapsack	budgeted selection, resource allocation	$O(n \cdot C)$
BFS/DFS	reachability, components, cycle detection	$O(V + E)$
Dijkstra	shortest paths, non-negative weights	$O((V + E) \log V)$
DSU	connectivity queries, Kruskal MST	near $O(1)$ amortized

Engineering pattern selection rule

- If your problem contains **a monotonic condition**, try binary search on answer.
- If your problem asks for **nearest boundary**, try monotonic stack.
- If your problem asks for **best next choice repeatedly**, try heap + greedy.
- If your problem asks for **best total under constraints**, try knapsack/DP.
- If your problem asks for **reachability**, start with BFS/DFS.

Reusable test checklist

```rust
pub fn test_checklist() {
    /* 1) empty input */
    /* 2) smallest valid input */
    /* 3) duplicates / equal values */
    /* 4) monotonic increasing and decreasing */
    /* 5) random small with brute-force cross-check */
    /* 6) large stress (performance / allocation) */
}
```

References

Foundational Algorithm Literature

This section lists academically established, widely used sources for the algorithmic foundations behind the patterns in this handbook.

- **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein**
  Introduction to Algorithms (MIT Press).
  A standard reference for asymptotic analysis, sorting, heaps, hash tables, graph algorithms, shortest paths, dynamic programming, and greedy design.

- **Robert Sedgewick, Kevin Wayne**
  Algorithms (Addison-Wesley).
  Strong focus on engineering practice: implementations, performance intuition, and real-world patterns.

- **Jon Kleinberg, Éva Tardos**
  Algorithm Design (Pearson).
  Excellent for rigorous design reasoning: greedy proofs, dynamic programming modeling, and graph algorithm correctness.

- **Steven S. Skiena**

The Algorithm Design Manual (Springer).
A pragmatic bridge between theory and engineering, with pattern recognition and problem mapping.

- **Donald E. Knuth**
  The Art of Computer Programming (Addison-Wesley), multiple volumes.
  Deep foundational treatment; especially valuable for understanding algorithms as disciplined computation rather than code recipes.

- **Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani**
  Algorithms (McGraw-Hill).
  Clean explanations for core paradigms: divide-and-conquer, greedy, dynamic programming, and graph methods.

- **Michael T. Goodrich, Roberto Tamassia**
  Algorithm Design and Applications / *Data Structures and Algorithms in Java* (concepts transfer directly).
  Solid for data structure fundamentals and algorithmic thinking.

- **Competitive programming references (pattern drilling)**
  Competitive Programming (Halim & Halim), *CP Handbook* (Antti Laaksonen).
  Useful as structured pattern practice catalogs; pair with the more rigorous sources above for proofs and correctness.

Rust Standard Library Documentation

The Rust ecosystem documentation below is considered primary, authoritative guidance for stable Rust usage and idioms in algorithm engineering. All material applies equally on Windows, Linux, and macOS.

- **The Rust Programming Language ("The Book")**
  Primary Rust language learning source: ownership, borrowing, lifetimes, traits, error handling, and standard collection usage.

- **The Rust Standard Library Documentation (`std`)**
  Authoritative reference for `Vec`, `VecDeque`, `HashMap`, `BTreeMap`, `BinaryHeap`, iterators, slices, strings, and algorithms like sorting and binary search patterns.

- **The Rust Reference**
  Language-level precision: expressions, ownership rules, moves, borrowing, pattern matching semantics, and trait resolution details.

- **Rust by Example**
  Short, working examples for common patterns that show correct ownership and borrowing in practical code.

- **The Cargo Book**
  Project structure, profiles, features, testing, benches, and build discipline required for reproducible engineering.

- **The Rustonomicon**
  For advanced boundaries: unsafe Rust invariants, aliasing rules, layout concerns, and performance-critical low-level details (useful even if this handbook remains mostly safe Rust).

- **Rust API Guidelines and Rust Style Guide**
  Conventions for naming, error models, and maintainability that matter when turning algorithm code into reusable library components.

Systems and Performance Engineering Sources

These sources explain how algorithms interact with real machines: caches, memory allocation, locality, branch prediction, and profiling methodology.

- **Randal E. Bryant, David R. O'Hallaron**
  Computer Systems: A Programmer's Perspective (CS:APP).
  A practical foundation for memory behavior, caching, CPU cost models, and performance reasoning.

- **Brendan Gregg**
  Systems Performance: Enterprise and the Cloud.
  A definitive guide to performance methodology: measurement, profiling, observability, and turning data into correct conclusions.

- **Ulrich Drepper**
  What Every Programmer Should Know About Memory.
  A classic deep dive into caches, locality, NUMA, and why memory behavior dominates performance in many real workloads.

- **Agner Fog**
  Software optimization manuals and microarchitecture notes.
  Highly practical for understanding pipeline effects, instruction-level performance, and how compilers and CPUs behave.

- **Intel and AMD architecture manuals**
  Vendor references for memory ordering, caching, and performance counters (useful when reasoning about low-level costs and benchmarking discipline).

- **Linux and Windows profiling documentation (conceptual)**

Even when code is cross-platform, performance measurement concepts (timers, sampling profilers, trace tools) translate across operating systems.

Graph Theory and Dynamic Programming References

This section focuses on graph modeling, shortest paths, connectivity, and DP design discipline.

Graph theory and graph algorithms

- **Reinhard Diestel**
  Graph Theory.
  A rigorous graph theory foundation: definitions, connectivity, cycles, and structural reasoning.

- **Douglas B. West**
  Introduction to Graph Theory.
  A strong teaching text for graph concepts that map directly to BFS/DFS reasoning.

- **J. A. Bondy, U. S. R. Murty**
  Graph Theory.
  A broad reference useful for deeper theoretical grounding beyond implementation.

- **Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin**
  Network Flows: Theory, Algorithms, and Applications.
  Excellent for shortest paths, flow networks, and practical algorithmic engineering around graphs.

Dynamic programming and optimization

- **CLRS (Cormen et al.)**

Strong DP chapters for modeling states, transitions, and correctness reasoning.

- **Kleinberg & Tardos**
  DP as disciplined design: recognizing optimal substructure and proving transitions.

- **Richard Bellman (foundational)**
  Original DP perspective: principle of optimality and staged decision processes (historical foundation, still conceptually central).

- **Knapsack and resource allocation modeling**
  Network flow and optimization texts often provide the cleanest viewpoint on knapsack-like constraints as engineering tradeoffs.

String algorithms

- **Dan Gusfield**
  Algorithms on Strings, Trees, and Sequences.
  A classic reference for string matching, prefix functions, and deeper text algorithms.

- **Foundational KMP sources (Knuth, Morris, Pratt)**
  Original formulation of KMP and the prefix function idea; useful to understand why the algorithm is linear and how its automaton emerges.

Engineering discipline note

- Use the Rust primary docs for the exact semantics and performance expectations of `Vec`, `HashMap`, `BTreeMap`, `BinaryHeap`, slices, iterators, and sorting.

- Use the algorithm texts to justify invariants, prove correctness, and choose the right paradigm.

- Use systems/performance references to ensure your algorithmic design remains fast on real hardware, not only in asymptotic notation.