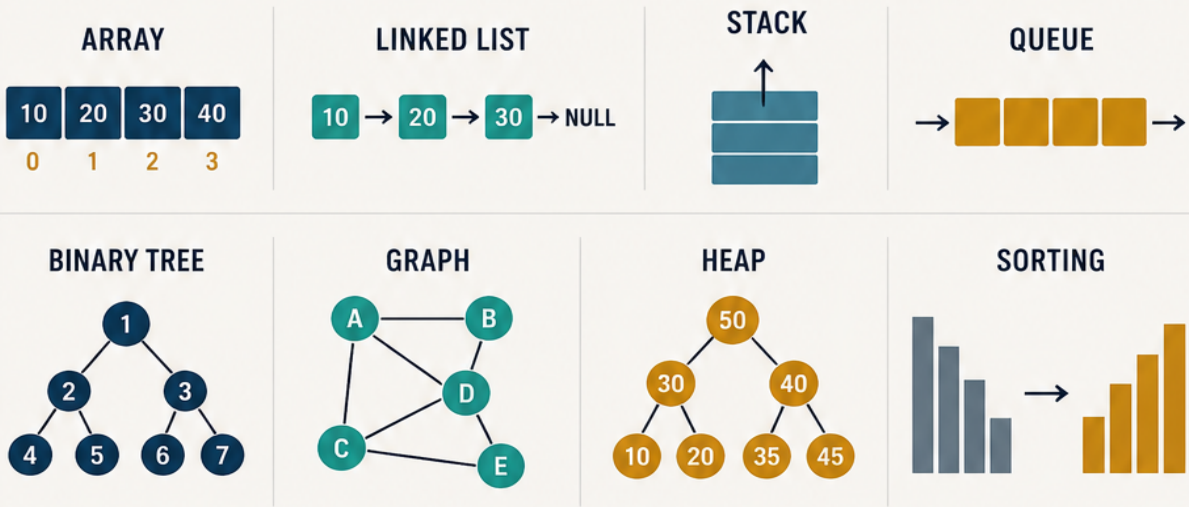
DATA STRUCTURES AND ALGORITHMS IN PROGRAMMING

A COMPLETE GUIDE FROM FUNDAMENTALS TO ADVANCED TECHNIQUES



JOHN TAO

Data Structures and Algorithms in Programming

A Complete Guide from Fundamentals to Advanced Techniques

Steve Publications

This book is available at
https://leanpub.com/datastructuresandalgorithmsinprogramming

This version was published on 2026-07-27

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Complete Guide from Fundamentals to Advanced Techniques

This book takes you from absolute beginner to professional mastery of data structures and algorithms. You will learn the theory, the implementation details, the trade-offs, and the real-world applications that matter in production software. Every concept is explained clearly with complete, working code examples in Python. No exercises or quizzes here; instead, you get thorough explanations, comparative analysis, and practical understanding that you can rely on as both a learning resource and a long-term reference.

Introduction: Why Data Structures and Algorithms Matter

Every software engineer eventually faces the same question. You have data that needs to be stored, searched, transformed, or transmitted. How do you organize it? The answer determines whether your application responds in milliseconds or minutes, whether it fits in memory or crashes under load, and whether future developers can maintain it or abandon it in frustration.

Data structures and algorithms are not abstract academic exercises reserved for computer science classrooms and technical interviews. They are the fundamental tools of software engineering, used every day in production systems that power search engines, databases, operating systems, compilers, network routers, recommendation engines, and virtually every non-trivial piece of software you interact with.

Consider a few concrete examples. When you type into a search box and see autocomplete suggestions appear instantly, a trie data structure is matching your prefix against millions of possible completions in microseconds. When you navigate using GPS, Dijkstra's algorithm or A* search is computing the shortest path through a graph with thousands of intersections. When a database executes a join between two tables, it uses hash tables, B-trees, or sort-merge operations to combine billions of rows efficiently. When your code editor highlights matching braces, a stack data structure tracks nesting depth as you type.

Understanding these tools is not about memorizing implementations. It is about developing an intuition for how different structures behave under different conditions, what trade-offs each one makes, and which patterns apply to which problems. A hash table gives you constant-time lookups but sacrifices ordering and iteration predictability. An array provides excellent cache performance and random access but expensive insertions in the middle. A balanced tree gives you ordered data with logarithmic operations but more complex implementation and higher memory overhead per node.

This book is designed to build that intuition systematically. We start from the very beginning, assuming no prior expertise beyond basic programming

concepts. Each chapter builds on previous material, introducing new structures and algorithms in a logical progression. By the time you reach the advanced chapters on dynamic programming, graph algorithms, and parallel computation, you will have developed the mental models needed to understand not just how these techniques work, but why they work and when to use them.

Every algorithm and data structure is presented with complete, production-quality Python code that you can run, modify, and study. The code follows modern best practices, includes proper error handling, and demonstrates real-world usage patterns rather than artificial toy examples. Where relevant, we compare multiple implementation approaches, analyze their time and space complexity, discuss edge cases and common pitfalls, and explain the underlying theory that justifies the design choices.

The book covers fifteen chapters organized into four major sections. The first section establishes foundations, covering algorithmic thinking, complexity analysis, recursion, and basic linear structures like arrays, linked lists, stacks, and queues. The second section explores advanced data structures including hash tables, trees, heaps, graphs, and specialized structures like tries and segment trees. The third section presents core algorithmic paradigms: sorting, searching, divide-and-conquer, greedy algorithms, backtracking, and dynamic programming. The final section covers cutting-edge topics used in modern systems, including randomized algorithms, parallel computation, string matching, and computational geometry.

You can read this book cover to cover as a comprehensive course, or use it as a reference when you encounter a specific problem that requires a particular technique. Either way, the goal is the same: to give you a deep, practical understanding of data structures and algorithms that you can apply throughout your career as a software engineer.

Chapter 1: Foundations of Programming and Algorithmic Thinking

Before we can discuss specific data structures and algorithms, we need to establish a shared foundation. This chapter introduces the core concepts and mental models that underlie everything else in this book. We will define what an algorithm is, explore how computational thinking helps us break down complex problems, and review the basic programming constructs that every algorithm relies on.

What Is an Algorithm

An algorithm is a precise, step-by-step procedure for solving a well-defined problem or performing a computation. The word itself comes from al-Khwarizmi, a ninth-century Persian mathematician whose work on arithmetic and algebra laid the foundations for systematic problem-solving methods.

What distinguishes an algorithm from casual instructions or vague guidelines is precision and determinism. An algorithm must specify exactly what to do at each step, with no ambiguity about which operation to perform next or when to stop. Given the same input, a deterministic algorithm always produces the same output and follows the same sequence of steps.

Consider a simple real-world example: a recipe for baking cookies. A good recipe is essentially an algorithm. It specifies exact quantities of ingredients (the input), precise operations like mixing, kneading, and baking (the steps), and clear termination conditions like "bake until golden brown" or "bake for 12 minutes." If you follow the recipe exactly, you should get cookies every time.

In programming, algorithms are expressed as sequences of instructions that a computer can execute. These instructions manipulate data, make decisions based on conditions, repeat operations in loops, and call other algorithms as subroutines. The quality of an algorithm is measured by several criteria:

- Correctness: Does it solve the problem for all valid inputs?

- Efficiency: How much time and memory does it use as input size grows?
- Clarity: Is it easy to understand, implement, and maintain?
- Robustness: Does it handle edge cases and invalid inputs gracefully?

These criteria often conflict. A more efficient algorithm might be harder to understand. Adding robustness checks slows execution slightly. Good algorithm design is about balancing these competing concerns based on the requirements of your specific application.

Let us look at a concrete example. Suppose we want to find the maximum value in a list of numbers. Here is a simple algorithm expressed in Python:

```
def find_maximum(numbers):
    """Find the maximum value in a non-empty list of numbers."""
    if not numbers:
        raise ValueError("Cannot find maximum of an empty list")

    max_value = numbers[0]  # Start with first element as candidate

    for num in numbers[1:]:  # Check each remaining element
        if num > max_value:
            max_value = num  # Update candidate when we find a larger value

    return max_value
```

This algorithm is correct because it examines every element and keeps track of the largest one seen so far. It runs in O(n) time, where n is the number of elements, because it makes exactly one pass through the list. It handles the edge case of an empty list by raising a clear error rather than returning a meaningless result.

The beauty of this simple algorithm is that it illustrates several fundamental principles. First, it uses a single variable to maintain state across iterations: max_value represents our best answer so far. Second, it processes input incrementally, making one comparison at a time. Third, it has a clear termination condition: when all elements have been examined, the current maximum must be the overall maximum.

These principles recur throughout algorithm design, whether we are sorting millions of records, routing packets through a network, or training a machine learning model. The specific operations change, but the underlying patterns remain remarkably consistent.

Computational Thinking and Problem Decomposition

Computational thinking is a problem-solving approach that breaks complex problems into manageable components, identifies patterns, abstracts away irrelevant details, and designs step-by-step solutions that can be executed systematically. It is not limited to computer science; it is a general intellectual toolkit applicable to any domain that requires structured reasoning.

Four pillars support computational thinking: decomposition, pattern recognition, abstraction, and algorithm design.

Decomposition means breaking a large problem into smaller, more tractable subproblems. Each subproblem can then be solved independently or in coordination with others. For example, building a web application decomposes naturally into frontend interface, backend logic, database schema, authentication system, and deployment pipeline. Each of these can be further decomposed: the backend might split into API endpoints, business logic, and data access layers.

Pattern recognition means identifying similarities between the current problem and problems you have solved before. If you have implemented a search feature for products, you can apply similar techniques to search for users, documents, or transactions. Recognizing patterns accelerates problem-solving because you can adapt existing solutions rather than starting from scratch each time.

Abstraction means focusing on the essential aspects of a problem while ignoring details that do not affect the solution. When designing a sorting algorithm, you care about comparing elements and rearranging them, but you do not care whether those elements are integers, strings, or customer objects. Abstraction allows algorithms to be general and reusable across different domains.

Algorithm design means creating a precise, executable procedure that solves the problem using the decomposed components and recognized patterns. This is where we translate our understanding into concrete instructions that a computer can follow.

Let us apply these four pillars to a practical problem: designing a system that recommends movies to users based on their viewing history.

First, decomposition. We can break this into subproblems:

- How do we represent user preferences and movie attributes?
- How do we measure similarity between users or between movies?
- How do we generate a ranked list of recommendations from similarity scores?
- How do we update recommendations efficiently as users watch new movies?

Second, pattern recognition. This problem resembles information retrieval, collaborative filtering, and nearest-neighbor search. We have solved similar problems in document ranking, friend suggestions, and product recommendations. The underlying mathematical structure is the same: given a query and a collection of items, rank items by relevance to the query.

Third, abstraction. We can represent both users and movies as vectors in a high-dimensional space, where each dimension corresponds to a feature like genre, director, actor, or rating. Similarity becomes a geometric concept: users with similar tastes have preference vectors that point in similar directions. This abstraction lets us use well-understood techniques from linear algebra and geometry, regardless of whether the features are explicit tags or learned embeddings.

Fourth, algorithm design. We can now specify concrete steps:

1. For each user, construct a preference vector from their viewing history and ratings.
2. For each movie, construct a feature vector from its attributes.
3. Compute similarity scores between the user's preference vector and each movie's feature vector using dot product or cosine similarity.
4. Sort movies by similarity score and return the top k recommendations, excluding movies already watched.

This decomposition reveals which data structures and algorithms are relevant. We need efficient vector storage and computation, fast similarity search (which suggests approximate nearest-neighbor techniques), and priority queues for ranking. The problem that initially seemed amorphous has become a concrete engineering task with well-defined components.

Variables, Types, and Memory at a High Level

To understand how algorithms manipulate data, we need a basic mental model of how programs represent and store information in computer memory.

At the lowest level, computer memory is a vast array of bits, grouped into bytes (eight bits each). Every piece of data your program uses, whether it is a number, a character, an image, or a complex object, is ultimately stored as some arrangement of these bits. Programming languages provide abstractions that let us work with meaningful types like integers, floating-point numbers, strings, and objects without worrying about the underlying bit patterns.

Variables are named references to values stored in memory. When you write `x = 42`, you are creating a variable named x that refers to the integer value 42. In languages like Python, variables are essentially labels attached to objects in memory; multiple variables can refer to the same object, and reassigning a variable simply changes which object it points to.

```
x = [1, 2, 3]
y = x        # y refers to the same list object as x
y.append(4)
print(x)     # Output: [1, 2, 3, 4]: modifying through y affects x too
```

This behavior, known as mutability, is crucial to understanding how algorithms work. When an algorithm modifies data in place, it changes the underlying object that variables refer to. When it creates new data, it allocates additional memory. The choice between in-place modification and creating new objects affects both time complexity (avoiding copies saves time) and space complexity (copies consume extra memory).

Data types determine what operations are valid on a value and how much memory it requires. Common primitive types include:

- Integers: whole numbers like 42, -17, or 0. In Python, integers have arbitrary precision and grow as needed; in languages like Java or C++, they have fixed sizes (32-bit or 64-bit).
- Floating-point numbers: approximate representations of real numbers like 3.14 or -0.001. They use a format that stores a significand and an exponent, allowing a wide range but limited precision.
- Booleans: logical values true or false, used for conditions and control flow.

- Characters and strings: sequences of Unicode code points representing text.

Composite types combine multiple values into structured units:

- Arrays and lists: ordered collections where elements are accessed by index.
- Tuples: fixed-size ordered collections, often used to group related values.
- Dictionaries and maps: key-value associations for fast lookup by key.
- Objects and structs: named fields holding related data, often with associated methods.

Understanding the memory layout of these types helps explain their performance characteristics. Arrays store elements contiguously in memory, which makes random access fast and iteration cache-friendly but insertion and deletion expensive because elements must be shifted. Linked lists store elements as separate nodes connected by pointers, which makes insertion and deletion cheap but random access slow and iteration less cache-efficient.

Control Flow: Sequencing, Selection, and Iteration

Every algorithm is built from three fundamental control flow constructs: sequencing, selection, and iteration. Together, these are sufficient to express any computable procedure, a result known as the structured programming theorem.

Sequencing means executing statements one after another in the order they appear. This is the simplest form of control: do step A, then step B, then step C. Most of the code you write relies on sequencing as its backbone.

Selection, also called branching or conditional execution, means choosing between different paths based on a condition. The most common form is the if-else statement:

```
def absolute_value(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x   # Path A: x is negative, negate it
    else:
        return x     # Path B: x is non-negative, return as-is
```

Selection allows algorithms to handle different cases, make decisions based on input values, and implement logic that depends on the current state. More complex forms include nested conditionals, multi-way branches (if-elif-else chains), and switch statements in some languages.

Iteration, also called looping, means repeating a block of code multiple times. There are two main types:

- Count-controlled loops repeat a known number of times, typically using a for loop that iterates over a range or collection.
- Condition-controlled loops repeat until a condition becomes false, typically using a while loop.

```
# Count-controlled: iterate over each element in a list
total = 0
for num in [1, 2, 3, 4, 5]:
    total += num

# Condition-controlled: repeat until sum exceeds threshold
total = 0
i = 0
while total < 100:
    i += 1
    total += i
```

Loops are where algorithms do most of their work. The number of times a loop executes, and what it does each time, directly determines the algorithm's time complexity. A single pass through n elements gives O(n) time. Nested loops, each running n times, give O(n^2) time. Loops that halve the remaining work each iteration, like binary search, give O(log n) time.

Understanding how control flow structures contribute to complexity is essential for analyzing and optimizing algorithms. We will explore this in depth in Chapter 2, but it is worth noting now that the structure of your loops often reveals the efficiency of your algorithm at a glance.

Functions, Modularity, and Abstraction

Functions, also called methods or procedures depending on the language, are named blocks of code that perform a specific task. They are the primary mechanism for organizing programs into manageable, reusable components.

Good function design follows several principles:

- Single responsibility: Each function should do one thing and do it well. A function named `find_maximum` should find the maximum value, not also validate input, log results, and send notifications.
- Clear interface: The function's name, parameters, and return value should make its purpose and usage obvious without reading the implementation.
- Minimal side effects: Functions should primarily compute and return values rather than modifying external state, which makes them easier to reason about, test, and compose.

Here is an example that demonstrates these principles:

```
def binary_search(sorted_list, target):
    """
    Find the index of target in sorted_list using binary search.

    Args:
        sorted_list: A list sorted in ascending order.
        target: The value to search for.

    Returns:
        The index of target if found, or -1 if not present.
    """
    left = 0
    right = len(sorted_list) - 1

    while left <= right:
        mid = (left + right) // 2

        if sorted_list[mid] == target:
            return mid
        elif sorted_list[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

This function has a clear responsibility (binary search), a well-defined interface (takes a sorted list and a target, returns an index), and no side effects (it does not modify the input list or any external state). You can call it from anywhere in your program, confident that it will do exactly what its name and documentation promise.

Modularity, the practice of organizing code into functions, classes, and modules, is closely related to abstraction. Abstraction means hiding implementation details behind a clean interface so that users of a component do not need to understand how it works internally, only what it does. When you call Python's built-in `sorted()` function, you do not need to know that it uses Timsort under the hood; you just need to know that it returns a new sorted list.

This principle scales up from individual functions to entire libraries and frameworks. A database library abstracts away file I/O, indexing, and query optimization so that application code can focus on business logic. An operating system abstracts away hardware details so that programs can run on different machines without modification. Abstraction is what makes large-scale software engineering possible.

Pseudocode vs. Real Code: Bridging the Gap

Pseudocode is an informal, human-readable description of an algorithm that uses programming-like syntax without being tied to any specific language. It is commonly used in textbooks, research papers, and algorithm design discussions because it focuses on the logic rather than language-specific details like type declarations, memory management, or library imports.

Here is how binary search might be expressed in pseudocode:

```
Algorithm BinarySearch(A, target)
    Input:  A sorted array A, a value target
    Output: Index of target in A, or -1 if not found

    left <- 0
    right <- length(A) - 1

    while left <= right do
        mid <- floor((left + right) / 2)

        if A[mid] == target then
            return mid
        else if A[mid] < target then
            left <- mid + 1
        else
            right <- mid - 1

    return -1
```

Pseudocode has advantages and disadvantages. It is accessible to anyone who understands basic programming concepts, regardless of their preferred language. It avoids distractions like syntax errors, library quirks, or performance characteristics specific to one implementation. On the other hand, pseudocode can hide important details: how are ties broken? What happens with duplicate values? How does the algorithm behave on edge cases like empty input or single-element arrays?

This book uses real, executable Python code throughout because concrete implementations reveal details that pseudocode obscures. When you see actual code, you can test it, modify it, and observe its behavior on different inputs. You can measure its runtime, inspect its memory usage, and verify that edge cases are handled correctly. Real code forces precision that pseudocode often avoids.

That said, the algorithmic ideas we discuss are language-independent. The binary search algorithm is the same whether implemented in Python, Java, C++, Rust, or JavaScript; only the syntax and some implementation details change. As you read, focus on the underlying logic and structure of each algorithm. Once you understand the concept, translating it into your preferred language is straightforward.

Chapter 1 Summary

This chapter established the foundations for everything that follows. We defined algorithms as precise, step-by-step procedures and discussed the criteria that make an algorithm good: correctness, efficiency, clarity, and robustness. We explored computational thinking and its four pillars, showing how decomposition, pattern recognition, abstraction, and algorithm design work together to tackle complex problems.

We reviewed basic programming concepts including variables, types, memory, control flow, and functions, emphasizing how these building blocks support algorithm implementation. We distinguished pseudocode from real code and explained why this book uses concrete Python implementations throughout.

The key takeaway is that algorithms are not mysterious or domain-specific; they are systematic approaches to problem-solving that rely on fundamental patterns and structures. With these foundations in place, we are ready to begin analyzing how efficiently different algorithms perform, which is the subject of the next chapter.

Chapter 2: Complexity Analysis and Big-O Notation

Efficiency is the defining characteristic that separates good algorithms from bad ones. Two algorithms might solve the same problem correctly, but one might handle millions of inputs in seconds while the other takes hours. Understanding why requires the tools of complexity analysis, which let us reason about how an algorithm's resource usage grows as input size increases.

This chapter introduces Big-O notation and related concepts for describing time and space complexity. We will learn how to analyze algorithms systematically, understand the most common complexity classes, distinguish between best/worst/average cases, and apply amortized analysis to data structures with variable-cost operations. By the end of this chapter, you will be able to look at any algorithm and estimate its performance characteristics without running it.

Why Efficiency Matters

Let us start with a concrete example that shows why efficiency is not an academic concern but a practical necessity.

Suppose you are building a social media platform and need to find pairs of users who share interests. You have a list of n users, and for each pair, you compute a similarity score. A straightforward approach checks every possible pair:

```
def find_all_pairs_naive(users):
    """Check every pair of users for shared interests."""
    pairs = []
    n = len(users)

    for i in range(n):
        for j in range(i + 1, n):
            similarity = compute_similarity(users[i], users[j])
            if similarity > THRESHOLD:
                pairs.append((users[i].id, users[j].id))

    return pairs
```

This algorithm examines n * (n - 1) / 2 pairs, which is approximately $n^2 / 2$. We say it runs in O(n^2) time, or quadratic time. For 1,000 users, this means about 500,000 comparisons. For 10,000 users, about 50 million. For 1 million users, about 500 billion.

Now imagine your platform grows to 10 million users. The naive approach would need roughly 50 trillion comparisons. Even at one billion comparisons per second (which is optimistic), this takes about 50 seconds just to find matching pairs once. If you need to update recommendations every minute, the naive algorithm is simply infeasible.

The solution is not faster hardware; it is a better algorithm. Using techniques like indexing, locality-sensitive hashing, or approximate nearest-neighbor search, we can reduce the complexity from O(n^2) to something closer to O(n log n) or even O(n). At 10 million users, O(n log n) means roughly 240 million operations instead of 50 trillion, a speedup of about 200,000 times.

This is why complexity analysis matters. It lets us predict whether an algorithm will scale to the sizes we need before we invest time implementing it. It guides our choice between competing approaches. And it explains why certain problems are fundamentally hard: no matter how clever our implementation, some problems require examining exponentially many possibilities in the worst case.

Time Complexity and Running Time Models

Time complexity describes how an algorithm's running time grows as a function of input size. We do not measure time in seconds or milliseconds because those depend on hardware, compiler optimizations, and system load. Instead, we

count the number of elementary operations the algorithm performs: comparisons, arithmetic operations, memory accesses, and so on.

The key insight is that different operations take different amounts of time, but they all fall into a small set of categories with predictable relative costs. On modern CPUs:

- Simple arithmetic and logical operations (add, subtract, compare) take one or a few nanoseconds.
- Memory access to cache takes a few nanoseconds.
- Memory access to main RAM takes tens to hundreds of nanoseconds.
- Disk I/O takes microseconds to milliseconds.

For complexity analysis, we typically treat basic operations as taking constant time, O(1), because their cost does not grow with input size. This abstraction lets us focus on the structure of the algorithm rather than micro-optimizations.

Let us analyze a simple example: computing the sum of all elements in an array.

```
def array_sum(arr):
    """Compute the sum of all elements in arr."""
    total = 0
    for num in arr:
        total += num
    return total
```

For an array of size n, this algorithm performs:

- One initialization (total = 0)
- n iterations of the loop
- In each iteration: one addition and one memory access

The total number of operations is proportional to n. We write this as O(n), read as "big-O of n" or "linear time." The constant factors and lower-order terms are ignored because they become negligible for large n.

Why do we ignore constants? Suppose Algorithm A performs 10n operations and Algorithm B performs n operations. For small n, Algorithm B is clearly faster. But both are O(n), and as n grows, the ratio between their running times

stays constant at 10:1. Big-O notation captures the growth rate, not the exact running time. When comparing algorithms with different growth rates (like O(n) vs O(n^2)), the growth rate dominates for large inputs, making constant factors irrelevant.

Big-O, Big-Omega, and Big-Theta Notation

Big-O is the most commonly used asymptotic notation, but it is only one of three related notations that describe different types of bounds on an algorithm's running time.

Big-O (O) describes an upper bound. When we say an algorithm runs in O(f(n)) time, we mean that for sufficiently large n, the running time is at most proportional to f(n). More formally, T(n) is O(f(n)) if there exist positive constants c and n_0 such that T(n) <= c * f(n) for all n >= n_0.

Big-Omega (Omega) describes a lower bound. T(n) is Omega(f(n)) if the running time is at least proportional to f(n) for large n. Formally, there exist positive constants c and n_0 such that T(n) >= c * f(n) for all n >= n_0.

Big-Theta (Theta) describes a tight bound, meaning both an upper and lower bound. T(n) is Theta(f(n)) if the running time is proportional to f(n), neither significantly faster nor slower. Formally, T(n) is Theta(f(n)) if it is both O(f(n)) and Omega(f(n)).

In practice, computer scientists often use Big-O loosely to mean "approximately this growth rate," when they technically mean Big-Theta. When we say binary search runs in O(log n) time, we usually mean it always takes about log n steps, which is technically Theta(log n). The distinction matters most in formal proofs and when discussing worst-case vs average-case behavior.

Here is a summary of the three notations:

- O(f(n)): "at most": the algorithm will not be slower than this (upper bound)
- Omega(f(n)): "at least": the algorithm will not be faster than this (lower bound)
- Theta(f(n)): "exactly": the algorithm grows at this rate (tight bound)

When analyzing an algorithm, we typically want to establish:

- A worst-case upper bound using Big-O

- An average-case estimate, often using Big-O informally
- Sometimes a lower bound using Big-Omega to show the algorithm is optimal for the problem

Analyzing Loops, Conditionals, and Function Calls

The core skill of complexity analysis is looking at an algorithm's structure and determining its time complexity. Let us work through the most common patterns.

Single loops: A loop that iterates n times, performing constant-time work each iteration, runs in O(n) time.

```
def print_all(arr):
    for item in arr:          # Runs n times
        print(item)           # O(1) per iteration
# Total: O(n)
```

Nested loops: Nested loops multiply their iteration counts. Two nested loops each running n times give O(n^2). Three nested loops give O(n^3).

```
def compare_all_pairs(arr):
    n = len(arr)
    for i in range(n):        # Runs n times
        for j in range(n):    # Runs n times per outer iteration
            if arr[i] < arr[j]:
                swap(arr, i, j)
# Total: O(n^2)
```

Sequential operations: When operations are executed one after another, their complexities add. The overall complexity is dominated by the largest term.

```
def process_data(arr):
    # Step 1: sort: O(n log n)
    arr.sort()

    # Step 2: linear scan: O(n)
    for item in arr:
        if item < 0:
            remove_item(arr, item)

    # Step 3: constant work: O(1)
    print("Done")

# Total: O(n log n) + O(n) + O(1) = O(n log n)
```

Conditionals: The complexity of an if-else statement is the maximum of the complexities of its branches, because in the worst case we execute the slower branch.

```
def find_element(arr, target):
    if is_sorted(arr):        # O(n) to check
        return binary_search(arr, target)  # O(log n)
    else:
        return linear_search(arr, target)  # O(n)

# Worst case: O(n) for checking + O(n) for linear search = O(n)
```

Halving loops: A loop that halves the remaining work each iteration runs in O(log n) time. This pattern appears in binary search and many divide-and-conquer algorithms.

```
def count_halving_steps(n):
    steps = 0
    while n > 1:
        n = n // 2          # Halve n each iteration
        steps += 1
    return steps
# Runs approximately log_2(n) times, so O(log n)
```

Recursive algorithms: Recursive algorithms are analyzed by writing a recurrence relation that expresses the running time in terms of smaller inputs. For example, merge sort divides the problem in half, recursively sorts each half, and then merges them:

$$T(n) = 2 * T(n/2) + O(n)$$

This reads as: to solve a problem of size n, we do two recursive calls on problems of size n/2, plus O(n) work for merging. Solving this recurrence gives T(n) = O(n log n). We will learn systematic techniques for solving recurrences in Chapter 13.

Space Complexity and Memory Usage

Space complexity measures how much additional memory an algorithm uses as a function of input size, excluding the space needed to store the input itself. Like time complexity, we express it using Big-O notation.

An algorithm that uses a fixed amount of extra memory regardless of input size has O(1) space complexity, also called constant space. An algorithm that creates an array proportional to the input size has O(n) space complexity, or linear space.

```
def sum_in_place(arr):
    """O(1) extra space: only uses a single variable."""
    total = 0
    for num in arr:
        total += num
    return total

def create_doubled(arr):
    """O(n) extra space: creates a new array of the same size."""
    result = []
    for num in arr:
        result.append(num * 2)
    return result
```

Space complexity matters for several reasons. First, memory is finite; algorithms that use too much space will crash on large inputs with out-of-memory errors. Second, memory allocation and deallocation have overhead; minimizing allocations improves performance. Third, algorithms that work in place (modifying the input rather than creating copies) are often preferred when the input can be safely modified.

There is frequently a trade-off between time and space. An algorithm that uses extra memory for caching or indexing might run faster than one that recomputes values on demand. For example, sorting an array with merge sort requires O(n) extra space but guarantees O(n log n) time, while heap sort uses

O(1) extra space but has worse constant factors and cache behavior. Choosing between them depends on whether you prioritize memory efficiency or raw speed.

Recursion introduces another dimension: each recursive call adds a frame to the call stack, which stores local variables and return addresses. A recursive algorithm with depth d uses O(d) stack space. Deep recursion can cause stack overflow errors if the call stack exceeds its limit. This is why iterative implementations are often preferred for performance-critical code, and why tail recursion optimization (which reuses the current stack frame instead of creating a new one) is valuable in languages that support it.

Best Case, Worst Case, and Average Case Analysis

Algorithms often behave differently on different inputs of the same size. Best-case analysis considers the minimum running time over all possible inputs. Worst-case analysis considers the maximum. Average-case analysis considers the expected running time assuming some probability distribution over inputs.

Worst-case is the most commonly reported because it provides a guarantee: no matter what input you give the algorithm, it will not be slower than this. This is critical for real-time systems, safety-critical applications, and competitive programming where you must ensure your solution completes within the time limit.

Best-case is rarely useful on its own because it often corresponds to trivial inputs. A search algorithm's best case is finding the target at the first position, giving O(1) time. But relying on best-case behavior is dangerous; production systems must handle adversarial or pathological inputs.

Average-case is informative but requires assumptions about the input distribution. If we assume all permutations of an array are equally likely, quicksort's average running time is O(n log n). But if the input is already sorted and we always choose the first element as pivot, quicksort degrades to O(n^2). Understanding when average-case assumptions hold is part of algorithm design.

Let us examine quicksort, which illustrates all three cases:

```python
import random

def quicksort(arr):
    """Sort arr in place using randomized quicksort."""
    if len(arr) <= 1:
        return

    # Random pivot selection avoids worst case on sorted input
    pivot_index = random.randint(0, len(arr) - 1)
    pivot = arr[pivot_index]

    # Partition into three groups
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    # Recursively sort and combine (in-place modification)
    quicksort(left)
    quicksort(right)

    arr[:] = left + middle + right

# Best case: O(n log n): each partition splits evenly
# Worst case: O(n^2): each partition is maximally unbalanced
# Average case: O(n log n): random pivots give balanced splits with high
↪  probability
```

The random pivot selection is crucial. Without it, an adversary could construct inputs that trigger worst-case behavior. With randomization, the expected running time is O(n log n) regardless of input, which is the hallmark of a robust algorithm.

Amortized Analysis Basics

Some data structures have operations where individual steps can be expensive but are rare enough that the average cost over many operations is low. Amortized analysis captures this phenomenon by averaging the cost across a sequence of operations rather than analyzing each operation in isolation.

The classic example is a dynamic array, like Python's list or Java's ArrayList. Appending an element to the end usually takes O(1) time: just write the value into the next available slot. But when the array is full, we must allocate a new, larger array and copy all existing elements over, which takes O(n) time where n is the current size.

If we analyzed append as its worst case, we would say it runs in O(n) time. But that is misleading because the expensive resize happens rarely. With geometric growth (doubling the capacity each time), the amortized cost of append is O(1). Here is why:

Suppose we insert n elements into an initially empty dynamic array that doubles when full. Resizes happen when we reach sizes 1, 2, 4, 8, 16, ..., up to the largest power of 2 less than or equal to n. The total number of element copies across all resizes is:

1 + 2 + 4 + 8 + ... <= 2n

This is a geometric series that sums to at most 2n. Adding the n initial insertions, the total work is at most 3n, so the average (amortized) cost per insertion is at most 3, which is O(1).

Three methods are commonly used for amortized analysis:

- Aggregate method: Compute the total cost of a sequence of operations and divide by the number of operations.
- Accounting method: Assign an artificial "charge" to each operation; cheap operations save credits that pay for expensive ones.
- Potential method: Define a potential function that measures stored energy in the data structure; amortized cost equals actual cost plus change in potential.

All three methods give the same result but offer different intuitions. The aggregate method is simplest and most direct. The accounting method is useful when you can identify which operations fund which future costs. The potential method is the most general and works well for complex data structures like splay trees or Fibonacci heaps.

Amortized analysis also applies to hash tables with chaining or open addressing, where insertions are O(1) amortized despite occasional rehashing when the table grows. It applies to union-find with path compression, where find operations are nearly constant time amortized. Understanding amortized complexity is essential for correctly analyzing many important data structures.

Common Complexity Classes in Practice

Let us summarize the most important complexity classes you will encounter, ordered from fastest to slowest:

O(1): Constant time: The running time does not depend on input size. Examples include accessing an array element by index, inserting at the front of a linked list, or looking up a key in a hash table (average case). These are the gold standard for performance.

O(log n): Logarithmic time: The running time grows proportionally to the logarithm of input size. Each step reduces the problem by a constant factor. Examples include binary search, operations on balanced binary search trees, and heap operations. Logarithmic algorithms scale extremely well; searching through one billion items takes only about 30 comparisons.

O(n): Linear time: The running time grows proportionally to input size. We examine each element once. Examples include linear search, finding the maximum in an unsorted array, and single-pass data processing. Linear is often the best we can do when every element must be examined at least once.

O(n log n): Linearithmic time: Slightly worse than linear but still very efficient. This is the complexity of optimal comparison-based sorting algorithms like merge sort, heap sort, and randomized quicksort (average case). Many divide-and-conquer algorithms fall into this class.

O(n^2): Quadratic time: The running time grows with the square of input size. Nested loops over the same data typically produce quadratic behavior. Examples include bubble sort, insertion sort on reverse-sorted input, and comparing all pairs of elements. Quadratic algorithms become impractical for inputs larger than a few thousand elements.

O(n^3): Cubic time: Three nested loops, or algorithms like Floyd-Warshall for all-pairs shortest paths. Used only when the problem inherently requires examining triplets or when input sizes are small.

O(2^n): Exponential time: The running time doubles with each additional input element. Algorithms that enumerate all subsets or try all possibilities often have exponential complexity. These are feasible only for very small inputs (n <= 20 or so).

O(n!): Factorial time: The number of operations grows as n factorial. Algorithms that enumerate all permutations fall into this class. Even for n = 15, 15! is about 1.3 trillion, making factorial algorithms impractical beyond tiny inputs.

The practical guideline is: aim for O(1), O(log n), O(n), or O(n log n) whenever possible. If your algorithm is O(n^2) or worse, look for a better approach before

deploying it to production with large data. Exponential and factorial algorithms are acceptable only when the problem is provably hard (NP-hard) and inputs are guaranteed to be small.

Chapter 2 Summary

This chapter introduced the tools for analyzing algorithm efficiency. We learned that Big-O notation describes how running time grows with input size, ignoring constant factors that become irrelevant for large inputs. We distinguished between Big-O (upper bound), Big-Omega (lower bound), and Big-Theta (tight bound).

We practiced analyzing common code patterns including loops, nested loops, conditionals, and recursive calls. We discussed space complexity and the trade-offs between time and memory. We explored best-case, worst-case, and average-case analysis, emphasizing that worst-case guarantees are most important for production systems. We introduced amortized analysis for data structures where occasional expensive operations are paid for by many cheap ones.

With these tools in hand, you can now look at any algorithm and estimate its performance characteristics. The next chapter builds on this foundation by introducing recursion, a powerful technique for expressing algorithms that operate on hierarchical or self-similar structures.

Chapter 3: Recursion and Inductive Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

The Recursive Mindset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Base Cases and Recursive Steps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Tracing Recursion: Stack Frames and Call Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Mathematical Induction as a Reasoning Tool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Tail Recursion and Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Common Recursive Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

When to Use (and Avoid) Recursion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 3 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 4: Arrays, Strings, and Linear Storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Static vs. Dynamic Arrays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Memory Layout and Cache Locality

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Array Operations: Insertion, Deletion, Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Two-Pointer Technique and Sliding Window

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Strings as Specialized Arrays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

String Operations and Mutable vs. Immutable Designs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Common Pitfalls and Off-by-One Errors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 4 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 5: Linked Lists, Stacks, and Queues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Pointer-Based Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Singly Linked Lists: Implementation and Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Doubly Linked Lists and Bidirectional Traversal

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Circular Linked Lists

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Stacks: Array-Based vs. Linked Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Queues: Standard, Circular, and Priority Variants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Deques and Real-World Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 5 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 6: Hashing and Hash Tables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

The Hashing Idea

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Designing Good Hash Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Collision Resolution: Chaining vs. Open Addressing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Load Factor and Dynamic Resizing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Linear Probing, Quadratic Probing, Double Hashing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Performance Analysis and Degenerate Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Real-World Applications and Language Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 6 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 7: Trees and Binary Search Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Tree Terminology and Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Binary Trees and Their Properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Tree Traversals: Inorder, Preorder, Postorder, Level Order

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Binary Search Trees: Operations and Complexity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Self-Balancing Trees: AVL Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Red-Black Trees and Their Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Tree-Based Sets and Maps in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 7 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 8: Heaps and Priority Queues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

The Heap Property

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Binary Heap Implementation as an Array

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Insertion, Extraction, and Heapify

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Build-Heap and Heap Sort

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Priority Queues as Abstract Data Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

D-Heaps and Fibonacci Heaps Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Applications: Scheduling, Median Finding, Top-K Problems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 8 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 9: Graphs and Graph Algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Graph Terminology and Representations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Adjacency Matrices vs. Adjacency Lists

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Breadth-First Search and Shortest Unweighted Paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Depth-First Search and Its Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Dijkstra's Algorithm for Weighted Shortest Paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Bellman-Ford and Negative Edge Weights

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Floyd-Warshall for All-Pairs Shortest Paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Minimum Spanning Trees: Prim and Kruskal

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Topological Sorting and Dependency Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Connected Components and Union-Find

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 9 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 10: Advanced Tree Structures — Tries, Segment Trees, and More

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Trie Data Structure for String Storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Trie Operations: Insert, Search, Prefix Matching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Memory Optimization: Compressed Tries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Segment Trees for Range Queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Building and Updating Segment Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Lazy Propagation in Segment Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Suffix Structures Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Interval Trees and Geometric Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 10 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 11: Sorting Algorithms — Complete Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Sorting Problem Definition and Stability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Comparison-Based Lower Bounds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Bubble Sort, Selection Sort, Insertion Sort

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Merge Sort: Divide-and-Conquer in Action

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Quick Sort: Partitioning and Pivot Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Heap Sort: In-Place and Guaranteed Performance

Counting Sort, Radix Sort, Bucket Sort

Hybrid Sorting: Introsort and Timsort

Stable Sort vs Unstable Sort: When It Matters

Choosing the Right Sort for Your Use Case

Chapter 11 Summary

Chapter 12: Searching Algorithms and Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Linear Search and Its Variants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Binary Search: Implementation and Edge Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Iterative vs. Recursive Binary Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Lower Bound, Upper Bound, and Equality Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Interpolation and Exponential Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Searching in Sorted Matrices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Fractional Cascading for Multiple Searches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Practical Search Library Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 12 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 13: Algorithmic Paradigms — Divide-and-Conquer, Greedy, Backtracking

Algorithm Design Paradigms Overview

Divide-and-Conquer Strategy and Master Theorem

Classic Divide-and-Conquer: Merge Sort, Quick Select, Closest Pair

Greedy Algorithms and Optimal Substructure

Proving Greedy Correctness with Exchange Arguments

Backtracking: Systematic Search with Pruning

Branch and Bound for Optimization Problems

Chapter 13 Summary

Chapter 14: Dynamic Programming — From Basics to Advanced

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

What Makes a Problem Amenable to DP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Memoization vs. Tabulation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Defining States and Transitions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Classic 1D Problems: Fibonacci, Knapsack, LIS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

2D DP: Edit Distance, Longest Common Subsequence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

DP on Trees and Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Space Optimization Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Advanced Patterns: Digit DP, Interval DP, Bitmask DP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 14 Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Chapter 15: Advanced Topics — Randomized, Parallel, String, and Geometric Algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Randomized Algorithms and Probabilistic Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Quick Sort Randomization and Monte Carlo Methods

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Parallel Algorithms and Divide-and-Conquer Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

String Matching: KMP Algorithm

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Rabin-Karp Rolling Hash for Pattern Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

Computational Geometry Basics: Convex Hull, Line Intersection

Advanced Optimization: Simulated Annealing, Genetic Algorithms Overview

Chapter 15 Summary

Conclusion: Putting It All Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/datastructuresandalgorithmsinprogramming.