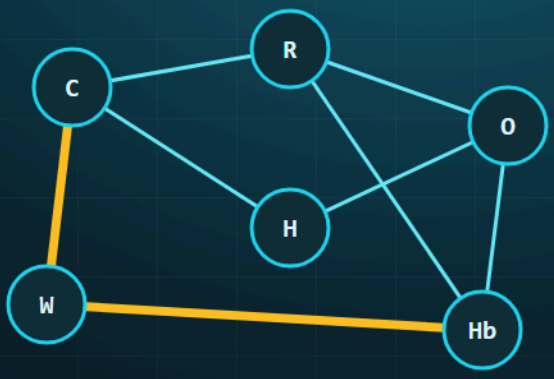
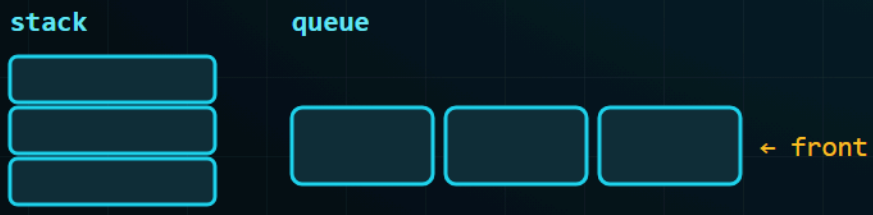




binary search tree



Central → Harbor: 17 min



O(1) · O(log n) · O(n log n)

recursion · DP · greedy

A FRIENDLY, VISUAL GUIDE

Algorithms and Data Structures for Beginners

Learn them step by step with Python

Big-O · arrays · linked lists · hash tables · sorting · recursion
trees · heaps · graphs · greedy · dynamic programming

PICTURES · TRACE TABLES · RUNNABLE CODE · PRACTICE

THENEXBOOKS

Algorithms and Data Structures for Beginners

A Friendly, Visual Guide with Python
From Big-O to Graphs and Dynamic Programming

First Edition

THENEXBOOKS

Algorithms and Data Structures for Beginners
A Friendly, Visual Guide with Python

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Timings depend on your computer and Python version; any timings shown are examples only.

Python is a trademark of the Python Software Foundation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by the Python Software Foundation.

Thistlewood Library, the running example in this book, is fictional, as are all books, authors, patrons, and places mentioned in its examples. Any resemblance to real libraries, books, or people is coincidental.

Preface

Every program you have ever used is built from two ingredients: **data**, and **steps** that do something with that data. A music app keeps your playlists; a map app keeps roads and finds a route; a search box keeps millions of words and finds the few you typed. How that data is organized — its **data structure** — and how the steps are arranged — the **algorithm** — decide whether the app answers instantly or makes you wait, whether it handles ten users or ten million.

Algorithms and data structures have a reputation for being hard: a wall of math, strange symbols, and puzzles that only make sense to people who already understand them. They do not need to be. Almost every idea in this book can be explained with something you already know — a stack of plates, a line at a counter, a phone's contact list, a paper map — and then turned into a short, readable Python program you can run and change.

That is how this book works. For each idea we start with a picture and an everyday example, walk through it step by step with a trace table so you can see exactly what happens, write the code in plain Python, and only then ask how fast it is and why. By the end you will not just recognize names like binary search, hash table, heap, or Dijkstra's algorithm — you will understand how they work, be able to write them yourself, and know when to use each one.

Who This Book Is For

- **Self-taught programmers and bootcamp graduates** who can write Python but never studied computer science.
- **Students** taking a first course in data structures and algorithms who want a clearer, friendlier explanation.
- **Developers in other fields** — data analysis, web, automation — who want to write faster, better-organized code.
- **Anyone curious** about how the programs they use every day actually work under the hood.

You should know basic Python: variables, `if` statements, `for` and `while` loops, functions, lists, and dictionaries. Chapter 2 is a quick refresher of everything else we use. No advanced math is required; when we need a little, we explain it.

What You Will Learn

- What algorithms and data structures are, and how to measure efficiency with Big-O notation — explained simply.
- A step-by-step approach to solving problems before writing code.
- Arrays, linked lists, stacks, queues, deques, hash tables, and working with strings.
- Linear and binary search; bubble, selection, insertion, merge, quick, counting, and radix sort.
- Recursion, explained with pictures and call stacks.

- Trees, binary search trees, balanced trees, heaps, and priority queues.
- Graphs, breadth-first and depth-first search, shortest paths, minimum spanning trees, topological sort, and union-find.
- Greedy algorithms, divide and conquer, backtracking, dynamic programming, and handy patterns such as tries, two pointers, and sliding windows.
- How to combine all of it into a real application.

How This Book Is Organized

Part I — Getting Started (Chapters 1–4) explains what algorithms and data structures are, refreshes the Python we need, introduces Big-O, and teaches a problem-solving method. **Part II — Linear Data Structures** (Chapters 5–10) covers arrays, linked lists, stacks, queues, hash tables, and strings. **Part III — Searching and Sorting** (Chapters 11–15) covers searching, sorting, and recursion. **Part IV — Trees and Heaps** (Chapters 16–19) introduces hierarchical data. **Part V — Graphs** (Chapters 20–23) covers networks and routes. **Part VI — Algorithm Design Techniques** (Chapters 24–28) teaches general strategies and ends with a complete case study. The appendices provide a Big-O cheat sheet, a Python data structure reference, a study plan, and a glossary.

Each chapter ends with a **Practice** section of exercises and a **Key Takeaways** box for quick review. Do the exercises — they are where the ideas really stick.

The Running Example: Thistlewood Library

Throughout the book we help the volunteers of **Thistlewood Library**, a fictional public library in a small town, build their catalog app. It starts with a handful of books in a Python list and grows chapter by chapter: searching and sorting the catalog, a fair queue for book holds, undo for librarians' edits, autocomplete in the search box, the fastest route for the delivery van between the library's six branches, and a planner for the summer reading challenge. Each new data structure or algorithm solves a real problem the library actually has, so you always know *why* you are learning it.

Conventions

Examples use Python 3.12 and only the standard library. We usually build each data structure ourselves first, so you can see how it works, and then show the built-in Python tool that does the same job in real code. Program output appears in blocks labeled **Output**, and trace tables show an algorithm's state step by step.

> **TIP**
>
> A helpful technique or a shortcut that makes your work easier.

NOTE

Background information, a deeper detail, or a useful aside.

WARNING

A common mistake that leads to wrong answers, crashes, or slow programs.

Contents

Preface

PART I — GETTING STARTED

1 What Algorithms and Data Structures Are, and Why They Matter

2 A Python Refresher for Algorithms: Lists, Loops, Functions, and Classes

3 Measuring Efficiency: Big-O Notation Explained Simply

4 Thinking Like an Algorithm Designer: Solving Problems Step by Step

PART II — LINEAR DATA STRUCTURES

5 Arrays and Dynamic Arrays

6 Linked Lists

7 Stacks

8 Queues and Deques

9 Hash Tables: How Dictionaries and Sets Work

10 Working with Strings

PART III — SEARCHING AND SORTING

11 Linear Search and Binary Search

12 Simple Sorts: Bubble, Selection, and Insertion Sort

13 Recursion Explained

14 Fast Sorts: Merge Sort and Quick Sort

15 Beyond Comparisons: Counting Sort, Radix Sort, and Choosing a Sort

PART IV — TREES AND HEAPS

16 Trees and Binary Trees

17 Binary Search Trees

18 Balanced Trees Without Fear: AVL Trees, Red-Black Trees, and B-Trees

19 Heaps and Priority Queues

PART V — GRAPHS

20 Graphs: Vocabulary and Representations

21 Graph Traversal: Breadth-First and Depth-First Search

22 Shortest Paths: Dijkstra's Algorithm and Friends

23 Minimum Spanning Trees, Topological Sort, and Union-Find

PART VI — ALGORITHM DESIGN TECHNIQUES

24 Greedy Algorithms

25 Divide and Conquer and Backtracking

26 Dynamic Programming for Beginners

27 Handy Patterns: Tries, Two Pointers, Sliding Windows, and Prefix Sums

28 Case Study: Building the Thistlewood Library Catalog

APPENDICES

A Big-O Cheat Sheet

B Python Data Structure Reference

C Study Plan and Practice Checklist

D Glossary

PART I

Getting Started

What algorithms and data structures are and why they matter, the Python you need, how to measure efficiency with Big-O notation, and a step-by-step method for solving problems before you write code.

1 · What Algorithms and Data Structures Are

2 · A Python Refresher

3 · Big-O Notation Explained Simply

4 · Thinking Like an Algorithm Designer

CHAPTER 1 · PART I — GETTING STARTED

What Algorithms and Data Structures Are, and Why They Matter

Every useful program does two things: it keeps information somewhere, and it follows steps to do something with that information. In this chapter you will learn what those two ideas are called, why choosing them well can make a program thousands of times faster, and how a small town library will help us explore them for the rest of the book.

If you have written a few Python programs, you have already used algorithms and data structures, even if nobody called them that. When you stored names in a list, you chose a data structure. When you wrote a `for` loop to find the largest number in that list, you wrote an algorithm. The words sound academic, but the ideas behind them are ordinary and practical. They are about organizing things sensibly and doing jobs in a smart order.

This book takes those everyday ideas and makes them precise. We will look at the classic ways programmers organize data, the classic step-by-step methods for searching, sorting, and exploring it, and simple ways to measure how fast those methods are. We will do it slowly, with pictures, with tables that trace every step, and with small Python programs you can run yourself. You do not need any computer science background. If you know variables, `if` statements, loops, functions, lists, and dictionaries, you are ready.

This first chapter is a map of the territory. We will define our two key terms, see a first example of why the choice of method matters, meet the library that serves as our running example, and explain how the rest of the book is organized.

What Is an Algorithm?

An **algorithm** is a finite list of clear, step-by-step instructions that takes some input and produces a result. That is the whole definition. A recipe is an algorithm for turning ingredients into a meal. The directions a friend gives you to their house are an algorithm for turning "where you are now" into "where they live". The method you learned at school for long division is an algorithm for turning two numbers into a quotient and a remainder.

Let us look at a tiny algorithm written in plain English. Imagine you are standing in front of a shelf of books and you want to know whether the library owns a book called *Tiny Machines*.

1. Start at the left end of the shelf.
2. Read the title of the book in front of you.
3. If it is *Tiny Machines*, stop: you found it.

4. Otherwise, if there is another book to the right, move one book to the right and go back to step 2.
5. If there are no more books, stop: the library does not have it on this shelf.

Nothing about this is clever, but it is a perfectly good algorithm. It has the properties that every algorithm needs:

- **Input.** It works on something given to it: a shelf of books and a title to look for.
- **Output.** It produces an answer: "found it" or "not here".
- **Definiteness.** Each step is clear. Anyone following the instructions does the same thing. "Read the title" leaves no room for guessing.
- **Finiteness.** It always stops. A shelf has a limited number of books, so after at most that many steps you are done.
- **Correctness.** When it stops, the answer is right. If the book is on the shelf, you will reach it. If you reach the end, it really was not there.

Those last two properties deserve a moment of attention. A set of instructions that might run forever is not an algorithm in the strict sense; it is a bug waiting to happen. And a set of instructions that stops quickly with the wrong answer is not useful, no matter how fast it is. Throughout this book, whenever we write an algorithm, we will ask two questions: *does it always stop?* and *is the answer always right?* Only then do we ask the third question: *how fast is it?*

Algorithms are not the same as code

An algorithm is an idea. Code is one way of writing that idea down so a computer can follow it. The shelf-search idea above could be written in Python, in JavaScript, in C, or on an index card for a new library volunteer. The idea stays the same. This is why computer scientists often describe algorithms in **pseudocode**, which is a relaxed, human-friendly mixture of English and programming notation that ignores the fussy details of any one language. We will use a little pseudocode in Chapter 4, but most of the time we will simply write Python, because Python is already close to pseudocode and you can run it.

Here is the same shelf search in Python. We store the titles in a list and walk through it one position at a time, counting how many titles we check along the way.

find_title.py

```
titles = [
    "The Clockmaker's Garden",
    "Rivers of Salt",
    "A Field Guide to Moths",
    "The Lantern Keeper",
    "Cooking with Fire",
    "Stars Over Thistlewood",
    "Tiny Machines",
    "The Quiet Orchard",
]
```

```
def find_title(titles: list[str], target: str) -> tuple[int, int]:
    """Return (position, steps). Position is -1 if the title is missing."""
    steps = 0
    for position in range(len(titles)):
        steps += 1
        if titles[position] == target:
            return position, steps
    return -1, steps


for wanted in ["Rivers of Salt", "The Quiet Orchard", "Moby-Dick"]:
    position, steps = find_title(titles, wanted)
    print(f"{wanted!r}: position {position}, checked {steps} titles")
```

Output

```
'Rivers of Salt': position 1, checked 2 titles
'The Quiet Orchard': position 7, checked 8 titles
'Moby-Dick': position -1, checked 8 titles
```

Notice three things in the output. When the book is near the front, we find it quickly. When it is at the very end, we check every title. When it is not there at all, we also check every title, because we cannot say "not here" until we have looked everywhere. That last case is important. With eight books it hardly matters, but a real library system can hold hundreds of thousands of records, and "check every one" starts to hurt. Keep this program in mind; we will improve on it later in the chapter and properly in Chapter 11.

> **NOTE**
>
> Python counts positions from 0, so the first title is at position 0 and the eighth is at position 7. Returning `-1` for "not found" is a common convention because `-1` can never be a real position in a list scanned from the front. Python's own `str.find` method uses the same trick.

What Is a Data Structure?

A **data structure** is a way of organizing data in a computer's memory so that certain jobs become easy. The key words are "so that certain jobs become easy". No single organization is best for everything. Each data structure makes some operations fast and others slow, and part of becoming a good programmer is learning which trade-off suits the job in front of you.

You already use data structures in daily life:

- A **stack of plates** lets you take the top plate instantly, but getting the bottom plate means lifting all the others first.
- A **queue at a bakery counter** serves people in the order they arrived. It is fair and simple, but you cannot easily jump to the fifth person.

- The **contacts app on your phone** keeps names in alphabetical order, so you can scroll to "P" and find a friend without reading every name.
- A **street map** records which places connect to which, and how far apart they are, so you can plan a route.
- A **family tree** arranges people in levels, each person linked to their parents and children.

Each of these has a direct computer equivalent, and each gets its own chapter: stacks (Chapter 7), queues (Chapter 8), sorted arrays and binary search (Chapters 5 and 11), graphs (Chapter 20), and trees (Chapter 16). Python even ships with several data structures built in. A Python `list` is a dynamic array. A `dict` is a hash table. A `set` is a hash table that stores only keys. You have been using professional-grade data structures since your first week of Python; this book shows you what is happening inside them and when to reach for something different.

Same data, different shapes

To see why organization matters, consider the same eight books stored in two different shapes. First, as a plain list of titles, in the order the volunteers typed them in:

```
index:   0                        1                 2  ...  7
       +------------------------+-----------------+-----+-----+------------------+
       | The Clockmaker's Garden | Rivers of Salt   | ... | ... | The Quiet Orchard |
       +------------------------+-----------------+-----+-----+------------------+
```

Second, as a dictionary that maps each book's ISBN to its title. An **ISBN** (International Standard Book Number) is a unique code printed on a book; the ones in our library are made up for this book.

```
  key (ISBN)             value (title)
  "978-0-00-000101"  -->  "The Clockmaker's Garden"
  "978-0-00-000102"  -->  "Rivers of Salt"
  ...
  "978-0-00-000108"  -->  "The Quiet Orchard"
```

The list is excellent if you want to print the books in the order they were entered, or grab "the third book". The dictionary is excellent if a patron hands you a book and you scan its ISBN, because a dictionary can jump straight to the entry for a key without looking at the others. Here is that lookup in action.

isbn_lookup.py

```
catalog = {
    "978-0-00-000101": "The Clockmaker's Garden",
    "978-0-00-000102": "Rivers of Salt",
    "978-0-00-000103": "A Field Guide to Moths",
    "978-0-00-000104": "The Lantern Keeper",
    "978-0-00-000105": "Cooking with Fire",
    "978-0-00-000106": "Stars Over Thistlewood",
    "978-0-00-000107": "Tiny Machines",
```

```
    "978-0-00-000108": "The Quiet Orchard",
}

print(catalog["978-0-00-000106"])
print("978-0-00-000999" in catalog)
print(catalog.get("978-0-00-000999", "not in the catalog"))
```

Output

```
Stars Over Thistlewood
False
not in the catalog
```

There is no loop in this program. We did not write "check the first key, then the second key". The dictionary finds the answer in roughly the same small amount of time whether it holds eight books or eight million. How it manages that trick is the subject of Chapter 9, and it is one of the most beautiful ideas in the whole subject. For now, the lesson is simpler: **the same data, arranged differently, makes different questions cheap**.

> **TIP**
>
> When you start a new program, ask yourself: "What questions will this program ask of its data most often?" If the answer is "look things up by a unique key", think of a dictionary. If it is "process everything in order", a list is often enough. If it is "always handle the most urgent item next", you will want a heap, which we meet in Chapter 19.

Algorithms and Data Structures Work Together

It is tempting to treat algorithms and data structures as two separate topics, but they are really two halves of one skill. A data structure is only useful because of the algorithms that operate on it, and many algorithms are only fast because of the data structure they use.

A good way to picture the relationship is a kitchen. The data structures are how the kitchen is organized: spices in a rack sorted alphabetically, pots hanging by size, a stack of clean plates by the sink. The algorithms are the recipes the cook follows. A great recipe in a chaotic kitchen is slow, because the cook spends all their time hunting for ingredients. A beautifully organized kitchen with a bad recipe still produces a bad meal. You need both.

In programming terms, here are a few pairings you will meet in this book:

Algorithm	Data structure it relies on	Chapter
Binary search (halve the range each step)	A sorted list	11
Undo and redo in an editor	Two stacks	7
Serving holds in arrival order	A queue (`deque`)	8

Always picking the most urgent task	A heap (priority queue)	19
Breadth-first search on a map	A graph plus a queue	21
Dijkstra's shortest-path algorithm	A graph plus a heap	22
Autocomplete as you type	A trie (prefix tree)	27

Do not worry if most of those names mean nothing yet. The point is that almost every famous algorithm is really "a clever idea plus the right way to store things". By the end of the book, every row of that table will feel familiar.

A First Taste of Efficiency: The Guessing Game

Let us see, with a game, why the choice of algorithm can matter enormously. A friend thinks of a whole number between 1 and 100. You guess, and after each guess your friend says "too low", "too high", or "correct". How should you guess?

One strategy is to guess 1, then 2, then 3, and so on. This is the same idea as our shelf search: check every possibility in order. If the secret number is 73, you need 73 guesses. If it is 100, you need 100.

A much better strategy is to always guess the middle of the range that is still possible. If the middle guess is too low, the secret must be in the upper half, so you throw away the whole lower half at once. If it is too high, you throw away the upper half. Each guess cuts the remaining possibilities roughly in half. This idea is called **halving**, and it is the heart of binary search.

guessing_game.py

```
def guess_number(secret: int, low: int, high: int) -> int:
    """Find secret in low..high by always guessing the middle. Return guess count."""
    guesses = 0
    while low <= high:
        guess = (low + high) // 2
        guesses += 1
        print(f"Guess {guesses}: range {low}..{high}, try {guess}")
        if guess == secret:
            return guesses
        if guess < secret:
            low = guess + 1       # too low: throw away the bottom half
        else:
            high = guess - 1      # too high: throw away the top half
    return guesses


total = guess_number(73, 1, 100)
print("Found it in", total, "guesses")
```

Output

```
Guess 1: range 1..100, try 50
Guess 2: range 51..100, try 75
Guess 3: range 51..74, try 62
Guess 4: range 63..74, try 68
Guess 5: range 69..74, try 71
Guess 6: range 72..74, try 73
Found it in 6 guesses
```

Six guesses instead of seventy-three. Let us trace exactly what happened, one row per guess. A **trace table** like this one records the values of the important variables at each step, and it is the single best tool for understanding (and debugging) an algorithm. You will see many of them in this book.

Guess	low	high	guess = (low + high) // 2	Answer	What changes
1	1	100	50	too low	low = 51
2	51	100	75	too high	high = 74
3	51	74	62	too low	low = 63
4	63	74	68	too low	low = 69
5	69	74	71	too low	low = 72
6	72	74	73	correct	stop

Look at the width of the range, `high - low + 1`, at the start of each guess: 100, then 50, then 24, then 12, then 6, then 3. It roughly halves every time. How many times can you halve 100 before only one number is left? About seven: 100, 50, 25, 13, 7, 4, 2, 1. So with the halving strategy you *never* need more than 7 guesses for any secret between 1 and 100, while counting up one at a time can need 100.

> **WARNING**
>
> The halving strategy only works because the numbers have an *order*. "Too low" tells you something only because every number is either smaller or larger than your guess. If your friend just said "wrong" without a hint, halving would be impossible and you would be back to trying every number. In the same way, binary search on a list of books only works if the list is sorted first. Forgetting this is one of the most common beginner mistakes, and we will return to it in Chapter 11.

Why Efficiency Matters

With 100 numbers, the difference between 7 guesses and 100 guesses is a minor annoyance. Now let us see what happens as the problem grows. The table below compares the worst case for the two strategies: checking one possibility at a time, versus halving each time. The halving column is simply "how many times can you halve *n* until one item is left", plus the final check.

Number of items (n)	One at a time (worst case)	Halving (worst case)
10	10	4
1,000	1,000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

Read that last row slowly. To find one item among a billion, the one-at-a-time method may need a billion steps. The halving method needs thirty. Even if your computer can do many millions of simple steps per second, a billion-step search done thousands of times a day adds up to real waiting. Thirty steps is effectively instant.

This is the central lesson of the whole subject: **for small inputs, almost any method is fine; for large inputs, the choice of method matters more than the speed of the computer.** A faster computer might make the slow method two or three times quicker. A better algorithm can make it thousands or millions of times quicker. No amount of new hardware closes a gap like a billion versus thirty.

Computer scientists have a compact way to describe this kind of growth, called **Big-O notation**. The one-at-a-time search is described as **O(n)**, read "order n", meaning the work grows in step with the number of items. The halving search is **O(log n)**, meaning the work grows with the number of times you can halve n, which grows very slowly. Chapter 3 explains Big-O from scratch, with plenty of counting and no scary mathematics. For now, just remember the shape of the table.

Efficiency is about more than speed

When programmers say an algorithm is efficient, they usually mean it is fast, but there is a second resource that matters: **memory**, also called **space**. Some algorithms go faster by using extra memory, for example by building a dictionary so that later lookups are instant. Others save memory by doing more work. Throughout this book, whenever we describe how an operation performs, we will give both its **time cost** (how many steps) and its **space cost** (how much extra memory). Often you will find that you can trade one for the other, and choosing wisely depends on the situation.

There is also a third kind of efficiency that is easy to forget: **your own time**. A simple algorithm that you understand completely, that is easy to test, and that is fast enough for the data you actually have is often better than a clever one that saves microseconds and hides bugs. Good programmers optimize when it matters, and they measure first to find out whether it does.

> **NOTE**
>
> Python's built-in tools are already very efficient. Sorting with `sorted()`, looking things up in a `dict` or `set`, and adding to the end of a `list` are all implemented carefully in C inside the Python interpreter. One goal of this book is to help you understand *why* they are fast, so you use them wisely, and to show you what to build when the built-ins are not enough.

Meet Thistlewood Library

Ideas stick better when they solve a real problem, so this book follows one project from start to finish. **Thistlewood Library** is a fictional public library system in the small town of Thistlewood. It has a main building called Central and five smaller branches: Riverside, Hilltop, Old Mill, Westgate, and Harbor. A delivery van carries books between them. A group of volunteers has started building a catalog app for the library in Python, and they would like our help to make it grow.

Here are the eight books that make up the library's sample catalog. We will use exactly these books throughout the book, so they will soon feel like old friends. All the books, authors, and ISBN-like codes are made up.

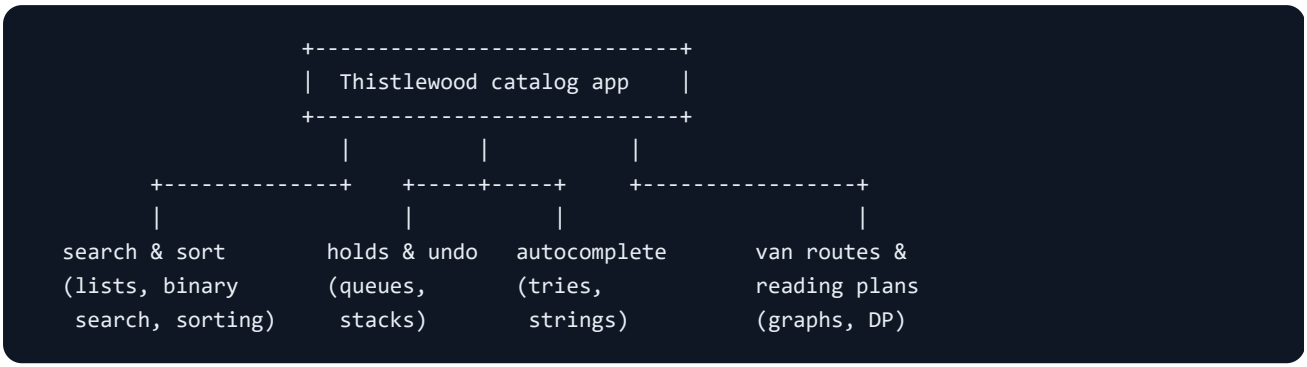
ISBN	Title	Author	Year	Pages	Genre
978-0-00-000101	The Clockmaker's Garden	Ada Winslow	2011	312	fiction
978-0-00-000102	Rivers of Salt	Tomas Reyes	1998	254	history
978-0-00-000103	A Field Guide to Moths	Priya Natarajan	2019	188	science
978-0-00-000104	The Lantern Keeper	Ada Winslow	2015	401	fiction
978-0-00-000105	Cooking with Fire	Marcus Bell	2021	220	cooking
978-0-00-000106	Stars Over Thistlewood	June Park	2008	176	local
978-0-00-000107	Tiny Machines	Leo Hartmann	2023	298	science
978-0-00-000108	The Quiet Orchard	Mei Tanaka	2005	342	fiction

The library also has patrons, the people who borrow books. Each patron has a library card with an ID. Our four sample patrons are Sam Ortiz (`TW-0042`), Nia Brooks (`TW-0107`), Omar Haddad (`TW-0215`), and Lucy Chen (`TW-0330`). *The Lantern Keeper* is so popular that three patrons have placed holds on it, and they must be served in the order they asked: first Nia (`TW-0107`), then Sam (`TW-0042`), then Lucy (`TW-0330`).

Eight books is a tiny catalog, and that is deliberate. With eight items you can trace every algorithm by hand, check every output with your own eyes, and see exactly what each line of code does. Whenever the size of the data matters, we will also think about what would happen with a catalog of a hundred thousand books, because that is where good algorithms earn their keep.

How the app grows through the book

The volunteers have a wish list. Almost every item on it turns out to be a classic algorithms problem in disguise:



Feature the volunteers want	Idea behind it	Chapters
Find a book by ISBN instantly	Hash tables (dictionaries)	9
List books by title, year, or length	Sorting	12, 14, 15
Find a title in a sorted shelf list	Binary search	11
Serve holds fairly, first come first served	Queues	8
Undo the last catalog edit	Stacks	7
Suggest titles as a patron types	Tries and string handling	10, 27
Keep the "most requested" list up to date	Heaps and priority queues	19
Plan the delivery van's route between branches	Graphs and shortest paths	20–23
Recommend books and plan reading challenges	Greedy methods, dynamic programming	24, 26

In Chapter 28 we put all these pieces together into one small but complete catalog program. By then you will have built every part yourself.

How This Book Works

This book has six parts. **Part I**, which you are reading now, gets you ready: a quick Python refresher tuned for algorithms (Chapter 2), a gentle introduction to Big-O notation (Chapter 3), and a step-by-step method for solving problems (Chapter 4). **Part II** covers the linear data structures, the ones that keep items in a line: arrays, linked lists, stacks, queues, hash tables, and strings. **Part III** is about searching and sorting, and includes a friendly introduction to recursion. **Part IV** climbs into trees and heaps. **Part V** explores graphs, which model anything made of connections, such as roads between library branches. **Part VI** teaches general design techniques, such as greedy algorithms and dynamic programming, and ends with the full Thistlewood case study.

Each chapter follows the same rhythm. First we build intuition with an everyday picture. Then we draw the idea with a diagram and walk through it with a trace table. Then we write it in Python, usually as a small class built from scratch. Then we measure its cost in time and space. Finally we compare our version with the tool Python already provides, so you know which to use in real programs.

Conventions used in this book

- **Code blocks** show complete, runnable Python unless the text says otherwise. A caption above a block, such as `find_title.py`, suggests a file name.
- **Output blocks**, captioned `Output`, show exactly what the program above them prints. If you type the code in and get something different, look closely: the difference is a clue.
- **Trace tables** show the values of variables step by step. Try to fill them in yourself before reading ours.
- **Tips** offer practical advice, **Notes** add background, and **Warnings** point out common traps.
- Each chapter ends with a **Practice** section of exercises and a **Key Takeaways** box that summarizes the chapter in a few lines.

Setting up

All the code in this book targets Python 3.12 and uses only the standard library, the modules that come with every Python installation. You do not need to install any extra packages. Most examples will also run on slightly older versions of Python 3, but if you are installing Python fresh, choose 3.12 or later. To check your version, open a terminal and run:

```
python --version
```

On some systems the command is `python3` instead of `python`. Any plain text editor works for writing the programs, and an editor with Python support, such as the free editors many beginners use, will color your code and catch typing mistakes. Save each example in a file, run it with `python filename.py`, and compare what you see with the Output block.

> **TIP**
> Do not just read the code; type it. Typing an algorithm yourself, making a small mistake, and fixing it teaches far more than reading a correct version. When a trace table appears, cover it with a sheet of paper and fill it in on your own first. Then check.

How to Study Algorithms Without Getting Lost

Many people find algorithms intimidating the first time, usually because they try to understand everything by staring at finished code. A few habits make it much easier.