Pseudocode & Algorithmic Problem-Solving

A Complete 10-Chapter Software Development Course

Topics covered:

Pseudocode conventions & structured programming · flowcharts

Problem decomposition & real-code translation

Brute force & greedy design strategies · divide and conquer

Recursive thinking & backtracking

Capstone: designing a conference room-scheduling algorithm from a real, ambiguous request

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Pseudocode & Algorithmic Thinking Matter for Programmers

Pseudocode & Algorithmic Problem-Solving

Chapter 1 · Why Pseudocode & Algorithmic Thinking Matter for Programmers

Pseudocode is a way of describing a solution's logic precisely, without committing to any one programming language's syntax. It sits between a plain-English problem statement and real code — precise enough that two different people (or the same person, translating into two different languages) should arrive at the *same* algorithm from it. This chapter opens with two real demonstrations of what happens when that precision slips: the same vague instruction, implemented two genuinely different but equally "correct" ways.

Demonstration 1: "Remove Duplicates" Isn't One Algorithm

A specification that says only "write an algorithm to remove duplicates from a list" sounds complete. It isn't — it never says whether the surviving elements should keep their original order.

```
# Interpretation A: put everything in a set
def dedupe_set(lst):
    return list(set(lst))

# Interpretation B: keep the first occurrence, preserve order
def dedupe_ordered(lst):
    seen = set()
    result = []
    for x in lst:
        if x not in seen:
            seen.add(x)
            result.append(x)
    return result
```

VERIFIED DIRECTLY — TWO GENUINELY DIFFERENT, EQUALLY "CORRECT" OUTPUTS

Run on `[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]`: Interpretation A returns `[1, 2, 3, 4, 5, 6, 9]`. Interpretation B returns `[3, 1, 4, 5, 9, 2, 6]`. Both genuinely contain each distinct value exactly once — both satisfy the plain-English spec completely — and they are **not the same list**. If a caller downstream assumed the original order was preserved

(say, to keep a shopping list in the order items were added), Interpretation A silently breaks that assumption while looking, by every reasonable test of "did it remove duplicates," perfectly correct.

Precise pseudocode forces this decision to be made *before* either version gets written: `ALGORITHM Dedupe(list) → preserve first-occurrence order` is now unambiguous, and any programmer translating it into any language will produce Interpretation B, not a coin flip between the two.

Demonstration 2: Loop Bounds Are a Real Off-By-One Trap

Pseudocode phrased as `for i = 1 to n` is standard textbook convention meaning **inclusive** of `n` — but a programmer translating it directly into a language whose native loop construct is exclusive of its upper bound can introduce a genuine bug without realizing the pseudocode said anything different.

VERIFIED DIRECTLY — THE SAME INTENDED LOOP, ONE REAL BUG

For `n=5`, the intended inclusive loop visits `[1, 2, 3, 4, 5]` — 5 iterations. Translating `for i = 1 to n` directly into Python's `range(1, n)` (exclusive of its stop value, a genuinely easy habit to reach for) visits only `[1, 2, 3, 4]` — **4 iterations, silently skipping `n` itself**. The pseudocode wasn't wrong; the translation dropped a piece of information the pseudocode's own convention had specified.

WHY THIS MATTERS MORE THAN IT LOOKS

This exact ambiguity is one of the most common real sources of off-by-one bugs — and it's entirely preventable at the pseudocode stage by being explicit: `for i = 1 to n inclusive`, or simply writing the loop's exact bound convention once, in one place, that every later translation can be checked against.

Five Concrete Situations Where This Skill Actually Gets Used

SITUATION	WHY PRECISE PSEUDOCODE MATTERS THERE
Technical interviews	Interviewers commonly ask for pseudocode or a verbal algorithm walkthrough before any code — ambiguity here reads as a lack of clarity, not a stylistic choice
Design docs before implementation	A design reviewed and approved in pseudocode form catches logic errors before a single line of real code — and real code — is written
Cross-team / cross-language specs	A shared algorithm description that doesn't commit to Python vs. Java vs. Go lets multiple teams implement the same logic independently and get matching results
Translating between languages	Porting an algorithm from one codebase's language to another goes through pseudocode as the language-neutral intermediate step, whether written down explicitly or not
Teaching and onboarding	A new team member unfamiliar with the codebase's language can still verify an algorithm's logic against clear pseudocode

What This Course Won't Cover

This course is deliberately about *designing* and *expressing* an algorithm, not about two things covered elsewhere on this site:

- **Formal complexity analysis** — Big-O notation, growth rates, and proving an algorithm's efficiency belong to Algorithms & Complexity (`algo1`); this course focuses on getting the logic right first, not on how fast it runs
- **Formal logic and proof techniques** — propositional/predicate logic, direct/contrapositive/induction proofs belong to Discrete Mathematics Fundamentals (`dmath1`); this course uses everyday conditional/loop logic without the formal proof machinery
- **Any single language's syntax in depth** — pseudocode is deliberately language-agnostic; Chapter 5 covers translation patterns across paradigms, not a specific language's own full feature set

> **WHY THIS SCOPE, SPECIFICALLY**
>
> Every topic from Chapter 2 onward builds toward one goal: taking a real, often ambiguous problem statement and arriving at a precise, translatable algorithm design — the skill this chapter's two demonstrations showed is genuinely easy to get subtly wrong.

Where This Course Is Headed

CHAPTER	TOPIC
2	Pseudocode Conventions & Structured Programming
3	Flowcharts & Visual Algorithm Representation
4	Problem Decomposition
5	Translating Pseudocode into Real Code
6	Brute Force & Exhaustive Search Strategies
7	Greedy Algorithms as a Design Strategy
8	Divide and Conquer as a Design Strategy
9	Recursive Thinking & Backtracking
10	Capstone — Designing an Algorithm from a Real-World Problem Statement

Hands-On Exercises

EXERCISE 1

Using this chapter's own two dedupe functions, run both on the list `["b", "a", "b", "c", "a"]` by hand (tracing through each function's own logic) and state what each one returns. Then write one precise sentence of pseudocode-style instruction that would have forced a single, unambiguous choice between them from the start.

View solution

EXERCISE 2

Using this chapter's own loop-bound finding, state how many times each of these would iterate, and whether it matches the "standard textbook inclusive" convention: (a) pseudocode `for i = 1 to 8`, translated as Python `range(1, 8)`; (b) the same pseudocode translated as Python `range(1, 9)`.

View solution

EXERCISE 3

Using this chapter's own five real-world situations table, pick the two situations you think are most different from each other in terms of *who* the pseudocode is actually being communicated to, and explain how that difference in audience might change how detailed or formal the pseudocode needs to be.

View solution

CHAPTER 1 QUICK REFERENCE

- **Pseudocode** describes an algorithm's logic precisely without committing to a specific language's syntax — the goal is that two different people translate it into the *same* underlying algorithm
- Verified: a vague "remove duplicates" spec produced two genuinely different, equally "correct" outputs — `[1,2,3,4,5,6,9]` vs. `[3,1,4,5,9,2,6]` — depending on an unstated order-preservation assumption
- Verified: `for i = 1 to n` translated naively into Python's exclusive `range(1, n)` silently drops the last iteration — 4 iterations instead of the intended 5, for `n=5`
- Five real situations where this matters: technical interviews, design docs, cross-team specs, language-to-language translation, teaching/onboarding
- Deliberately out of scope: formal complexity analysis (`algo1`), formal logic/proof (`dmath1`), deep single-language syntax
- **Next chapter:** Pseudocode conventions and structured programming — the actual notation and building blocks this course uses from here on

CHAPTER 2 OF 10

Pseudocode Conventions & Structured Programming

Pseudocode & Algorithmic Problem-Solving

Chapter 2 · Pseudocode Conventions & Structured Programming

Chapter 1 showed what goes wrong when an algorithm's description is vague. This chapter gives this course's own pseudocode a fixed, consistent notation, and covers the deeper reason that notation only needs three kinds of building blocks — sequence, selection, and iteration — to describe absolutely any algorithm, no matter how complex.

This Course's Pseudocode Notation

Every pseudocode block in this course uses the same conventions: keywords in `UPPERCASE` (`IF` / `THEN` / `ELSE` / `ENDIF`, `WHILE` / `ENDWHILE`, `FOR` / `ENDFOR`), `←` for assignment, and indentation to show nesting — deliberately close to what most textbooks and technical interviews already use, so nothing here needs relearning elsewhere.

```
ALGORITHM FindMax(list)
    max_val ← list[0]
    FOR i ← 1 TO length(list) - 1 // inclusive, per Chapter 1's own convention
        IF list[i] > max_val THEN
            max_val ← list[i]
        ENDIF
    ENDFOR
    RETURN max_val
```

VERIFIED DIRECTLY — THE PSEUDOCODE TRANSLATES TO WORKING CODE THAT MATCHES A TRUSTED REFERENCE

Translated directly into Python and run on `[3, 7, 2, 9, 4, 9, 1]`: the pseudocode's own logic returns `9`, exactly matching Python's own battle-tested built-in `max()` function on the identical input. The pseudocode wasn't just readable — it described a genuinely correct algorithm, confirmed against an independent, trusted implementation rather than just "looking right."

The Three Building Blocks

CONSTRUCT	WHAT IT DOES	THIS COURSE'S NOTATION
Sequence	Statements execute one after another, in the order written	Plain lines, top to bottom
Selection	Choose between different paths based on a condition	`IF...THEN...ELSE...ENDIF`
Iteration	Repeat a block of steps while a condition holds, or a fixed number of times	`WHILE...ENDWHILE`, `FOR...ENDFOR`

This isn't just a stylistic preference. It's backed by a real, formal result in computer science: the **Böhm–Jacopini theorem** (1966) proves that *any* computable function — however complex — can be expressed using only these three constructs, with no need for arbitrary jumps between arbitrary points in a program.

A Real, Verified Demonstration: No Goto Needed

Older languages (early BASIC, Fortran, assembly) relied heavily on `GOTO` — an unconditional jump to any labeled point in the program. To make the Böhm–Jacopini claim concrete rather than just cited, here's the same task — sum the even numbers from 1 to `n` — implemented two genuinely different ways: one using only sequence, selection, and iteration, and one built as an actual label-and-jump interpreter that mimics classic goto-driven control flow, with no `while` or `for` anywhere in its own execution logic.

```
# Structured: sequence, selection, iteration only
def sum_evens_structured(n):
    total = 0
    i = 1
    while i <= n:
        if i % 2 == 0:
            total = total + i
        i = i + 1
    return total

# Goto-simulated: a real label/jump interpreter, no while/for driving it
def sum_evens_goto(n):
    label = 'LOOP_START'
    i, total = 1, 0
    while label != 'DONE':  # the interpreter's own dispatch loop, not the algorithm's logic
        if label == 'LOOP_START':
            if i > n: label = 'END'; continue
            if i % 2 != 0: label = 'SKIP'; continue
            total = total + i; label = 'SKIP'; continue
        if label == 'SKIP': i = i + 1; label = 'LOOP_START'; continue
        if label == 'END': label = 'DONE'; continue
    return total
```

VERIFIED DIRECTLY — IDENTICAL RESULTS, FROM TWO STRUCTURALLY DIFFERENT CONTROL-FLOW STRATEGIES

Run on `n=20`: the structured version returns `110`. The goto-simulated version, jumping between labeled blocks `42` separate times to get there, also returns `110` — an exact match. Two genuinely different control-flow strategies, one real computed result.

IF THEY COMPUTE THE SAME THING, WHY PREFER STRUCTURED PROGRAMMING AT ALL?

Because `GOTO` lets execution jump to *any* labeled point from *anywhere*, reasoning about a large goto-driven program means tracking every possible jump into and out of every block — the number of paths through the code grows explosively as the program grows. Sequence, selection, and iteration each have exactly one entry point and one exit point, which is exactly why a structured program can be reasoned about (and later, in this course, formally decomposed and analyzed) one block at a time. This is precisely the argument Edsger Dijkstra made in his famous 1968 letter "*Go To Statement Considered Harmful*" — not that goto-based programs are wrong, but that they're needlessly hard to reason about compared to an equally capable structured alternative.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
This course's fixed pseudocode notation	Every worked example from Chapter 3 onward uses this exact notation without re-explaining it
Sequence/selection/iteration sufficiency, verified directly	Chapter 4's problem decomposition treats each subproblem as its own small structured block — a direct consequence of "one entry, one exit"
Structured programs being easier to reason about block by block	Chapter 9's recursive/backtracking designs lean on exactly this reasoning discipline to stay tractable

Hands-On Exercises

EXERCISE 1

Using this chapter's own `FindMax` pseudocode as a template, write pseudocode in this course's own notation for an algorithm that finds the *smallest* value in a list. Translate it into working code and verify it matches a trusted reference (e.g. Python's own `min()`) on the list `[8, 3, 5, 1, 9, 2]`.

View solution

EXERCISE 2

Using this chapter's own `sum_evens_goto` function as a guide, trace through what `label` the interpreter would be at after exactly 4 label-jumps when run with `n=3`, and state the final value of `total` it would return.

View solution

EXERCISE 3

Using this chapter's own explanation of why structured programming is preferred over goto (even though the Böhm–Jacopini theorem proves they're equally capable), explain in your own words the specific difference between "can compute the same thing" and "is equally easy to reason about" — and why a course on algorithmic problem-solving cares more about the second property than the first.

View solution

CHAPTER 2 QUICK REFERENCE

- **This course's pseudocode notation:** `UPPERCASE` keywords, `←` for assignment, indentation for nesting — verified translating faithfully to working code matching a trusted reference (`FindMax` matched Python's own `max()`)
- **Three building blocks:** sequence (order), selection (`IF/ELSE`), iteration (`WHILE` / `FOR`) — proven sufficient for any computable algorithm by the **Böhm–Jacopini theorem** (1966), no `GOTO` required
- Verified directly: a structured version and a real goto-simulated (label/jump) version of the same task produced the identical result, `110`, for summing even numbers 1 to 20 — confirming the theorem concretely, not just citing it
- Structured programming is preferred not because goto *can't* compute the same things, but because sequence/selection/iteration's "one entry, one exit" shape makes a program dramatically easier to reason about block by block — Dijkstra's own 1968 argument
- **Next chapter:** Flowcharts and visual algorithm representation — a second, visual notation for exactly the same three building blocks

CHAPTER 3 OF 10

Flowcharts & Visual Algorithm Representation

Pseudocode & Algorithmic Problem-Solving

Chapter 3 · Flowcharts & Visual Algorithm Representation

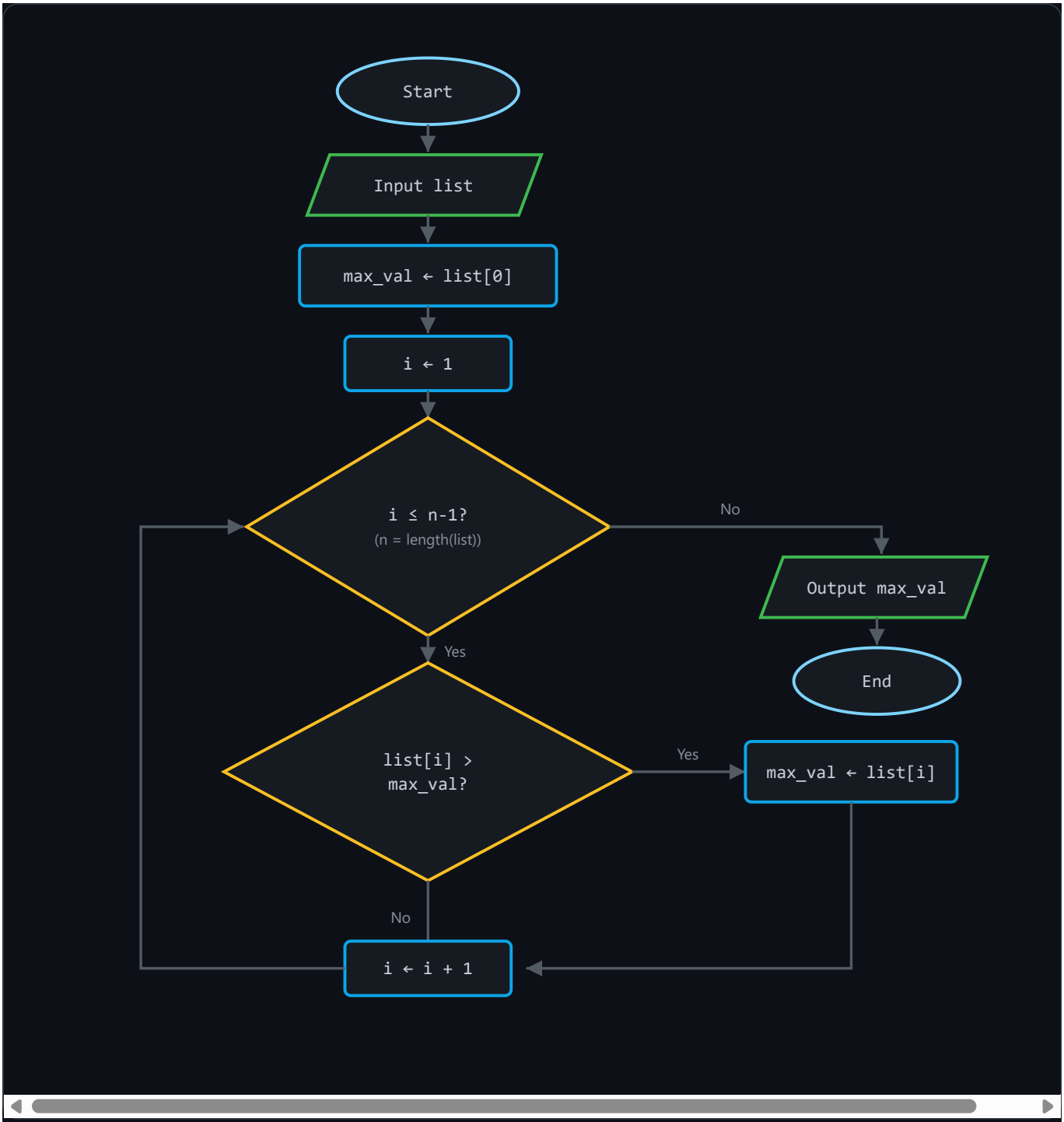
A flowchart is a second notation for the exact same three building blocks Chapter 2 covered — sequence, selection, iteration — drawn instead of written. It isn't a replacement for pseudocode; it's an alternative view of the identical logic, useful in some situations and genuinely counterproductive in others.

Standard Flowchart Symbols

SHAPE	MEANING	CORRESPONDS TO
Oval (terminal)	Start or end of the algorithm	The algorithm's own boundaries — not a pseudocode line at all
Parallelogram (I/O)	Input or output	`READ` / `DISPLAY` / `RETURN`
Rectangle (process)	A single computation or assignment	A sequence step, e.g. `max_val ← list[0]`
Diamond (decision)	A yes/no branch point	`IF...THEN...ELSE`
Arrow	Direction of flow — including looping back to an earlier point	`WHILE` / `FOR`'s own repeat behavior

Converting Pseudocode to a Flowchart — the Exact Same Algorithm as Chapter 2

Below is Chapter 2's own `FindMax` pseudocode, redrawn as a flowchart using only the five symbols above. Nothing about the underlying algorithm changed — only its notation.



Flowchart for FindMax(list) — every shape maps directly to one line of Chapter 2's own pseudocode

VERIFIED DIRECTLY — HAND-TRACING THE FLOWCHART MATCHES CHAPTER 2'S OWN ALREADY-VERIFIED RESULT

Tracing the flowchart by hand on `[3, 7, 2, 9, 4, 9, 1]` — the exact same test list Chapter 2 used — step by step through every diamond and arrow: `max_val` starts at `3`, updates to `7` at `i=1`, then to `9` at `i=3`, and never changes again (`9 > 9` is false at `i=5`, matching strict-greater-than exactly). The flowchart outputs `9` — identical to Chapter 2's own verified pseudocode-and-Python result on the same input. Two different notations, traced independently, agree exactly.

When a Flowchart Helps

- **Non-technical stakeholders** — a shape-and-arrow diagram is often easier for someone who doesn't read code fluently to follow than the equivalent pseudocode
- **A small number of branches** — a handful of decision points renders as a compact, genuinely clarifying picture
- **Spotting a missing case at a glance** — an arrow that visibly goes nowhere, or a decision with no "else" path drawn, is often easier to notice in a diagram than scanning indentation in text

When a Flowchart Hurts — Demonstrated by the Diagram Above

> **LOOK AT WHAT IT TOOK TO DRAW SEVEN LINES OF PSEUDOCODE**
>
> Chapter 2's own `FindMax` pseudocode is seven lines long. Its flowchart above needed eleven shapes, a loop-back arrow routed all the way around the left side of the diagram, and careful spacing just to keep the arrows from crossing each other illegibly. This is the real, concrete version of a well-known critique of flowcharts: **they don't scale**. A loop that's a single line of pseudocode (`WHILE...ENDWHILE`) becomes a physically large loop-back arrow spanning the whole diagram — and a real algorithm with several nested loops and more than two or three decision points quickly turns into a tangle that's harder to follow than the pseudocode it was meant to clarify.

Flowcharts also handle two things this course will need soon particularly badly: **recursion** (a flowchart has no natural way to show a process "calling a smaller copy of itself," the subject of Chapter 9) and **complex data structures** (a single box like `max_val ← list[0]` hides everything about how `list` is actually organized — fine for a simple list, unworkable for a tree or graph).

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Pseudocode and flowchart verified to represent identical logic	Confirms both notations are interchangeable views of the same design — Chapter 5's translation-to-real-code discipline applies equally to either starting point
The loop-back arrow's own visual size, demonstrated concretely	A direct, visual argument for why this course uses pseudocode as its primary notation from Chapter 4 onward, reserving flowcharts for the situations Chapter 3 identified they genuinely help
Flowcharts handling recursion poorly	Sets up Chapter 9's own recursive framing, which needs pseudocode's own call-based notation instead

Hands-On Exercises

EXERCISE 1

Using this chapter's own flowchart, hand-trace it on the list `[5, 5, 5]` and state the final output. Then explain, using this chapter's own decision-diamond wording (`list[i] > max_val?`, strictly greater than), why the output is correct even though every element is identical.

View solution

EXERCISE 2

Using this chapter's own symbol table, draw (in words — describe each shape and its label in order) a simple three-shape flowchart for the pseudocode `IF x < 0 THEN DISPLAY "negative" ELSE DISPLAY "non-negative" ENDIF`, including a Start and End terminal.

View solution

EXERCISE 3

Using this chapter's own "when a flowchart hurts" argument, explain why an algorithm with three nested loops (a loop inside a loop inside a loop) would be a particularly bad candidate for flowchart representation, connecting your answer to what happened to just one loop in this chapter's own `FindMax` diagram.

View solution

CHAPTER 3 QUICK REFERENCE

- **Five standard symbols:** oval (start/end), parallelogram (input/output), rectangle (process), diamond (decision), arrow (flow, including loop-back)
- Verified directly: hand-tracing the `FindMax` flowchart on the same test list Chapter 2 used produces the identical result, `9` — pseudocode and flowchart are interchangeable views of the same logic
- Flowcharts help most for non-technical audiences, a small number of branches, and spotting a visibly missing case
- Verified concretely: seven lines of pseudocode required eleven shapes and a full-diagram loop-back arrow — flowcharts genuinely don't scale to larger algorithms, and handle recursion and complex data structures particularly badly
- **Next chapter:** Problem decomposition — breaking a large, ambiguous problem into the kind of small, well-defined subproblems this chapter's own single-block-per-step diagrams assumed from the start

CHAPTER 4 OF 10

Problem Decomposition

Pseudocode & Algorithmic Problem-Solving

Chapter 4 · Problem Decomposition

A real problem statement rarely arrives in a form small enough to write pseudocode for directly. **Top-down design** is the discipline of splitting a large, ambiguous problem into smaller subproblems — and splitting those, if needed — until each remaining piece is simple enough that Chapter 2's own three building blocks can express it directly.

Input / Processing / Output: A Starting Discipline

Before decomposing anything, frame the problem in three parts — what goes in, what has to happen to it, and what comes out. This alone often reveals the natural first split.

IPO	FOR THIS CHAPTER'S PROBLEM: "COMPUTE A SHOPPING CART'S FINAL TOTAL"
Input	A list of cart items, each with a quantity and unit price
Processing	Subtotal → discount tier → shipping → tax → sum — genuinely several distinct steps, not one
Output	A single final total the customer pays

The "Processing" row is doing too much to write pseudocode for directly — that's the signal to decompose it further.

Top-Down Decomposition, One Level

Splitting "Processing" into its natural major steps produces five subproblems, each independently understandable without needing to know how the others are implemented internally:

WHY STOP AT EXACTLY FIVE, AND NOT FEWER OR MORE

Each of these five boxes is now small enough to write pseudocode for directly — Chapter 5's own translation discipline can turn any one of them into real code without needing to think about the other four at the same time. That's the actual stopping rule for top-down decomposition: not a fixed number of levels, but "can I now write straightforward pseudocode for this piece, using only sequence, selection, and iteration?"

Pseudocode for Each Subproblem — and a Full, Verified Composition

Each of the five boxes above becomes its own short block of pseudocode, using a concrete cart to ground the numbers: 3 Widgets at $12.50, 1 Gadget at $45.00, 2 Gizmos at $8.25. Discount tiers: 10% off at $100+, 5% off at $50+. Free shipping at $75+ (discounted), otherwise a flat $8. Tax: 8%.

```
// Subproblem 1
ALGORITHM CalculateSubtotal(items)
    subtotal ← 0
    FOR EACH (name, qty, price) IN items
        subtotal ← subtotal + (qty × price)
    ENDFOR
    RETURN subtotal

// Subproblem 2
ALGORITHM ApplyDiscountTier(subtotal)
    IF subtotal ≥ 100 THEN rate ← 0.10
    ELSE IF subtotal ≥ 50 THEN rate ← 0.05
    ELSE rate ← 0.00
    ENDIF
    RETURN subtotal × (1 - rate)

// Subproblem 3
ALGORITHM CalculateShipping(discounted_subtotal)
    IF discounted_subtotal ≥ 75 THEN RETURN 0
    ELSE RETURN 8
    ENDIF

// Subproblem 4
ALGORITHM CalculateTax(discounted_subtotal)
    RETURN discounted_subtotal × 0.08

// Subproblem 5
ALGORITHM ComputeCartTotal(items)
    subtotal ← CalculateSubtotal(items)
    discounted ← ApplyDiscountTier(subtotal)
    shipping ← CalculateShipping(discounted)
    tax ← CalculateTax(discounted)
    RETURN discounted + shipping + tax
```

VERIFIED DIRECTLY — RUNNING THE COMPOSED SUBPROBLEMS END TO END

Translated into real code and run on the worked cart: `CalculateSubtotal` returns `99.00` (`3×12.50 + 1×45.00 + 2×8.25`). `ApplyDiscountTier` selects the `5%` tier (since `50 ≤ 99 < 100`), giving `94.05` . `CalculateShipping` returns `0` (free — `94.05 ≥ 75`). `CalculateTax` returns `7.52` (`94.05 × 0.08` , rounded). `ComputeCartTotal` sums these to a final total of `101.57` — matching a hand-computed reference calculation exactly.

DECOMPOSITION CHANGES STRUCTURE, NEVER THE ANSWER

The four sub-algorithms compute exactly what one large, undivided block of code would compute for the same cart — decomposition didn't add or remove any logic, it only organized it into pieces that can each be written, tested, and verified independently. If `CalculateShipping` alone had a bug, only that one subproblem's own pseudocode needs re-checking, not the entire calculation.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
IPO framing revealing where a problem needs further splitting	The default first move Chapter 10's own capstone applies to a genuinely ambiguous problem statement
Five independently verifiable subproblems, composed and confirmed end to end	Chapter 5's translation discipline works subproblem by subproblem — exactly this chapter's own decomposition, not the whole algorithm at once
"Small enough to write pseudocode for directly" as the real stopping rule	Chapters 6-8's design strategies (brute force, greedy, divide and conquer) all operate on problems already broken down to roughly this size

Hands-On Exercises

EXERCISE 1

Using this chapter's own five sub-algorithms, compute the final total for a cart containing just 1 item: a single $120.00 item, quantity 1. Show the result of each of the five subproblems in order, the same way this chapter's own worked example did.

View solution

EXERCISE 2

A new requirement arrives: orders over $200 (after discount) get a $5 shipping discount even if shipping wasn't already free. Using this chapter's own decomposition, identify which single subproblem needs to change, and explain why the other four don't need to be touched at all.

View solution

EXERCISE 3

Using this chapter's own IPO framing and stopping rule, apply the same discipline to a new problem: "given a list of student test scores, compute each student's letter grade and the class average." Write the Input/Processing/Output table, and list what you'd expect the first-level decomposition's major subproblems to be.

View solution

CHAPTER 4 QUICK REFERENCE

- **Input/Processing/Output (IPO)** framing is the starting discipline — an overloaded "Processing" row is the signal a problem needs decomposing
- **Top-down design:** split a problem into major subproblems, and split further only until each piece is small enough for direct pseudocode — not a fixed number of levels
- Verified directly: five independently-defined subproblems (subtotal, discount tier, shipping, tax, sum), composed and run end to end, produced `101.57` for a worked cart — matching a hand-computed reference exactly
- Decomposition reorganizes logic, it never changes the answer — and it lets any one subproblem be checked or fixed without touching the others
- **Next chapter:** Translating pseudocode into real code — taking exactly this kind of decomposed design and mapping it onto a specific language and paradigm

CHAPTER 5 OF 10

Translating Pseudocode into Real Code

Pseudocode & Algorithmic Problem-Solving

Chapter 5 · Translating Pseudocode into Real Code

Pseudocode describes *what* an algorithm does. Real code has to also decide *how* that logic is organized — as a sequence of steps, as an object's own behavior, or as a chain of transformations. The same pseudocode can translate correctly into genuinely different-looking real code, and a few specific translation habits are a real, verified source of bugs when the target language doesn't behave quite the way the source pseudocode implicitly assumed.

One Pseudocode Algorithm, Three Paradigms

```
ALGORITHM SumOfSquaresOfEvens(list)
    total ← 0
    FOR EACH x IN list
        IF x MOD 2 = 0 THEN
            total ← total + (x × x)
        ENDIF
    ENDFOR
    RETURN total
```

```python
# 1. Imperative/procedural — a direct, line-by-line translation
def sum_squares_evens_imperative(lst):
    total = 0
    for x in lst:
        if x % 2 == 0:
            total = total + (x * x)
    return total

# 2. Object-oriented — the same logic, wrapped as an object's own behavior
class EvenSquareSummer:
    def __init__(self, lst):
        self.lst = lst
    def compute(self):
        total = 0
        for x in self.lst:
            if x % 2 == 0:
                total += x * x
        return total

# 3. Functional — the same logic, as a composed chain of transformations
def sum_squares_evens_functional(lst):
    return sum(map(lambda x: x*x, filter(lambda x: x % 2 == 0, lst)))
```

VERIFIED DIRECTLY — THREE GENUINELY DIFFERENT-LOOKING TRANSLATIONS, ONE IDENTICAL ANSWER

Run on `[1, 2, 3, 4, 5, 6, 7, 8]`: the imperative version returns `120`, the OOP version's `.compute()` returns `120`, and the functional version returns `120` — an exact match across all three, confirming that paradigm is a choice about *how code is organized*, not a change to the algorithm the pseudocode actually described.

PARADIGM	WHAT CHANGES	WHAT STAYS THE SAME
Imperative/procedural	State (`total`) is mutated directly, step by step	Reads almost line-for-line like the pseudocode itself
Object-oriented	The list and the operation on it are bundled into one object	The internal loop logic is identical to the imperative version
Functional	No mutable variable at all — `filter` and `map` build new sequences, `sum` combines them	Still visits every element and applies the identical condition and computation

A Real, Verified Translation Pitfall: Truthiness Isn't Universal

Pseudocode often writes `IF list IS EMPTY THEN ...`. A common shortcut when translating this is to rely on the target language's own "truthiness" rules instead of checking length explicitly — a habit that works in some languages and silently breaks in others.

```python
# Python: naive translation of "IF list IS EMPTY"
def check_empty_naive(lst):
    if not lst:
        return 'EMPTY - handled'
    return 'not empty'
```

```javascript
// JavaScript: the "same" naive translation
function checkEmptyNaive(list) {
  if (!list) {
    return 'EMPTY - handled';
  }
  return 'not empty (or naive check failed to detect empty)';
}
```

VERIFIED DIRECTLY — THE SAME-LOOKING CHECK, CORRECT IN ONE LANGUAGE, SILENTLY BROKEN IN ANOTHER

Run on an empty list `[]`: Python's `not lst` correctly reports `'EMPTY - handled'`. The *line-for-line equivalent* JavaScript, `!list`, reports `'not empty (or naive check failed to detect empty)'` — because in JavaScript, **every array is truthy, including an empty one**. An empty array and a non-empty array both fail to trigger the naive check, and the code gives no indication anything went wrong.

THE FIX — AND WHY IT GENERALIZES

The correct, portable translation of `IF list IS EMPTY` checks length explicitly — `len(lst) == 0` in Python, `list.length === 0` in JavaScript — rather than leaning on whatever a given language happens to consider "falsy." This exact pitfall generalizes beyond empty lists: integer division is another classic example — many mainstream languages (Java, C, C++, C#, Go) default `/` between two integers to *integer* division, silently truncating, while Python 3's `/` always produces a float (`//` is Python's explicit integer-division operator). Pseudocode's own `÷` or `/` doesn't specify which behavior is intended — the translator has to decide, explicitly, every time.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Three paradigms, one verified-identical algorithm	Chapters 6-9's own design strategies are described in pseudocode precisely so they translate cleanly into whichever paradigm a real codebase already uses
A verified, language-specific truthiness gotcha	A concrete instance of Chapter 1's own general warning: an algorithm's own precision doesn't automatically survive translation without deliberate, explicit choices
Functional-style `filter` / `map` composition	The same "process every element, combine the results" shape Chapter 6's brute-force strategies apply directly

Hands-On Exercises

EXERCISE 1

Using this chapter's own three paradigm translations as a template, write an OOP-style translation for the pseudocode `ALGORITHM CountVowels(word)`, which counts how many of the letters in `word` are vowels (a, e, i, o, u). Verify your class produces the correct count for the word `"algorithm"`.

View solution

EXERCISE 2

Using this chapter's own verified truthiness finding, explain what would happen if the naive JavaScript `checkEmptyNaive` function were called on the string `""` (an empty string) instead of an empty array, and why this case behaves differently from the empty-array case.

View solution

EXERCISE 3

Pseudocode contains the line `average ← total / count`. Using this chapter's own integer-division discussion, explain what a programmer translating this into a C-family language (where `total` and `count` are both declared as integers) needs to do differently to get the same result as the equivalent Python 3 code, and why simply copying the pseudocode's `/` symbol directly is not guaranteed to be correct.

View solution

CHAPTER 5 QUICK REFERENCE

- Verified directly: the same `SumOfSquaresOfEvens` pseudocode, translated into imperative, OOP, and functional Python, produced the identical result (`120`) across all three — paradigm changes organization, not the algorithm
- Verified directly: a naive empty-list check (`not lst` / `!list`) works correctly in Python but silently fails in JavaScript, because every array is truthy in JavaScript, even an empty one
- The portable fix: check length explicitly (`len(lst)==0`, `list.length===0`) rather than relying on a language's own truthiness rules
- Integer division is a second classic pitfall: Java/C/C++/C#/Go default `/` between integers to truncating integer division; Python 3's `/` is always a float (`//` is its explicit integer-division operator) — pseudocode's own `/` doesn't specify which is intended

- **Next chapter:** Brute force and exhaustive search — the first of four design strategies, expressed in pseudocode ready to translate into any of this chapter's own three paradigms

CHAPTER 6 OF 10

Brute Force & Exhaustive Search Strategies

Pseudocode & Algorithmic Problem-Solving

Chapter 6 · Brute Force & Exhaustive Search Strategies

Once a problem has been decomposed to a subproblem small enough for direct pseudocode (Chapter 4), the very first design question is often: *could I just try every possibility?* **Brute force** — systematically checking every candidate solution until the right one turns up — is a genuine, often-correct first strategy, not a lazy fallback. This chapter shows exactly where that's true, and exactly where it stops being true, with real measured numbers rather than a rule of thumb.

Worked Example 1: Two Sum, Checked Exhaustively

```
ALGORITHM TwoSumBruteForce(list, target)
    n ← length(list)
    FOR i ← 0 TO n-1
        FOR j ← i+1 TO n-1
            IF list[i] + list[j] = target THEN
                RETURN (i, j)
            ENDIF
        ENDFOR
    ENDFOR
    RETURN NOT_FOUND
```

VERIFIED DIRECTLY

Run on `[2, 7, 11, 15, 3, 6]` looking for a pair summing to `9`: the algorithm checks every pair in order and returns indices `(0, 1)` — `list[0]+list[1] = 2+7 = 9`, confirmed correct. No cleverness was needed; every possible pair was simply checked in turn until a match appeared.

Worked Example 2: A Small-Keyspace PIN Search

A 3-digit PIN has exactly `1,000` possible values (`000` through `999`). Checking every single one is a completely legitimate approach when the space really is this small.

VERIFIED DIRECTLY

Exhaustively generating and checking every 3-digit combination against a correct PIN of `482`: the search found it after `483` attempts, in `0.187 milliseconds`. That's a measured rate of roughly **2.58 million attempts per second** for this simple a check — fast enough that "just try everything" isn't a compromise here; it's the obviously correct, simplest thing to do.

When Brute Force Is the Honest Answer

- **The search space is genuinely small** — hundreds or thousands of possibilities, as verified above, finish in a fraction of a second
- **Correctness matters more than speed, and the deadline allows it** — an exhaustive check is trivially easy to convince yourself (and a reviewer) is correct, since it never skips a case by construction
- **It's a legitimate first draft** — a working brute-force solution, even one that will later be replaced by Chapters 7 or 8's own smarter strategies, gives a correct reference answer to test a faster version against

When Brute Force Stops Being Honest — A Real, Measured Limit

Consider a genuinely different problem: given a set of `n` items, check every possible subset (for example, to find one that sums to a target value). The number of subsets of a set of size `n` is `2ⁿ` — and unlike the PIN search's fixed 1,000 candidates, this grows with the input itself.

VERIFIED DIRECTLY — THE RAW SUBSET COUNTS

`n=10`: `1,024` subsets. `n=20`: `1,048,576`. `n=30`: `1,073,741,824` — over a billion. `n=40`: `1,099,511,627,776` — over a *trillion*.

EXTRAPOLATED FROM THIS CHAPTER'S OWN MEASURED RATE — NOT A THEORETICAL ESTIMATE

At the exact same `≈2.58 million checks/second` rate the PIN search actually achieved above: checking all `2²⁰` subsets would take `≈0.41 seconds` — still fine. Checking all `2³⁰` subsets would take `≈6.9 minutes` — noticeably worse, but tolerable. Checking all `2⁴⁰` subsets would take `≈4.93 days` — for a set of only **40 items**, at a rate this chapter itself already measured as fast. This isn't a hypothetical slowdown; it's the same exhaustive-search strategy, the same measured speed, applied to a problem whose candidate count genuinely explodes as the input grows.

This is the honest boundary: brute force is the right answer exactly as long as the number of candidates stays small *relative to how fast they can be checked* — and the PIN search and the subset search differ only in how their candidate count behaves as the input grows, not in how the strategy itself works.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Brute force verified correct and fast on two small worked examples	The baseline every later design strategy in this course is honestly compared against, not dismissed outright
The measured 2.58M/sec rate extrapolated to a real explosion	Directly motivates Chapter 7's greedy strategy and Chapter 8's divide and conquer — both exist specifically to avoid checking every possibility
"Small relative to how fast candidates can be checked" as the real boundary	Algorithms & Complexity's own formal Big-O treatment gives this exact intuition a precise mathematical name, for anyone continuing on to that course

Hands-On Exercises

EXERCISE 1

Using this chapter's own `TwoSumBruteForce` pseudocode, hand-trace it on `[4, 1, 8, 3]` looking for a pair summing to `11`. List every pair the algorithm actually checks, in order, until it finds a match (or exhausts all pairs).

View solution

EXERCISE 2

Using this chapter's own measured PIN-search rate (≈2.58 million checks/second), estimate roughly how long an exhaustive search of every 6-digit PIN (1,000,000 possibilities) would take at that same rate, and explain why this is still a reasonable brute-force candidate even though it's 1,000 times larger than the 3-digit search.

View solution

EXERCISE 3

Using this chapter's own distinction between the PIN search and the subset search, explain in your own words why "the search space is 1,000 possibilities" and "the search space is 2^{10} possibilities" describe the exact same number (1,024 vs. 1,000, close enough), yet one of these problems stays brute-forceable as its input grows and the other doesn't.

View solution

CHAPTER 6 QUICK REFERENCE

- **Brute force / exhaustive search:** systematically check every candidate until a match is found — verified correct on Two Sum (`[2,7,11,15,3,6]`, target `9` → indices `(0,1)`) and a 3-digit PIN search (found in `483` attempts, `0.187ms`, `≈2.58M checks/sec`)

- Brute force is the honest choice when the candidate count is genuinely small relative to how fast candidates can be checked — not a lazy fallback
- Verified directly: subset counts explode as `2ⁿ` — `1,024` at `n=10`, over a **trillion** at `n=40`
- Extrapolated from this chapter's own measured rate: checking all subsets of just 40 items would take `≈4.93 days` — the same strategy, the same real speed, a fundamentally different-shaped problem
- **Next chapter:** Greedy algorithms — the first design strategy that avoids checking every possibility, and an honest look at where that shortcut can go wrong

CHAPTER 7 OF 10

Greedy Algorithms as a Design Strategy

Pseudocode & Algorithmic Problem-Solving

Chapter 7 · Greedy Algorithms as a Design Strategy

A **greedy** algorithm makes the choice that looks best *right now*, at every step, and never revisits or reconsiders it. It's fast and simple to write — and whether it's actually correct depends entirely on the problem's own structure. This chapter runs the exact same greedy algorithm on two different inputs: one where it provably matches the true optimum, and one where it provably doesn't.

The Algorithm: Greedy Coin Change

```
ALGORITHM GreedyCoinChange(denominations, amount)
    SORT denominations DESCENDING
    coins_used ← empty list
    remaining ← amount
    FOR EACH d IN denominations
        WHILE remaining ≥ d
            ADD d TO coins_used
            remaining ← remaining - d
        ENDWHILE
    ENDFOR
    RETURN coins_used
```

At every step, it grabs the largest coin that still fits — the locally optimal choice — and never looks back.

Case 1: Greedy Matches the True Optimum

Run on standard US coin denominations (`25, 10, 5, 1`), and cross-checked against a true optimum computed independently (via dynamic programming — trying every combination systematically, not greedily):

VERIFIED DIRECTLY — GREEDY MATCHES THE TRUE OPTIMUM EXACTLY, THREE TIMES

`amount=41` : greedy gives `[25,10,5,1]` , 4 coins — the independently-computed true optimum is also 4 coins.
`amount=63` : greedy gives `[25,25,10,1,1,1]` , 6 coins — true optimum is also 6. `amount=99` : greedy gives

`[25,25,25,10,10,1,1,1,1]`, 9 coins — true optimum is also 9. Three separate amounts, greedy's result matches an independently-verified optimum every time.

Case 2: The Exact Same Algorithm, Verified to Fail

Run the identical `GreedyCoinChange` pseudocode — nothing about the algorithm itself changes — on a different denomination set: `{1, 3, 4}`, making change for `6`.

VERIFIED DIRECTLY — A REAL, MEASURED GREEDY FAILURE

Greedy picks the biggest coin that fits first: `4` (remaining `2`), then `1` (remaining `1`), then `1` (remaining `0`) — `[4, 1, 1]`, **3 coins**. Independently computing the true optimum (again via dynamic programming, checking every combination systematically): `[3, 3]`, **2 coins**. Greedy's locally-best first choice — take the `4` — actively prevented it from reaching the genuinely better 2-coin answer.

WHY GRABBING THE "OBVIOUSLY BEST" COIN BACKFIRES HERE

Taking the `4` first leaves a remainder of `2` — and with denominations `{1,3,4}`, `2` can only be made from two `1`-coins, since there's no `2`-coin and no way to use a `3` or a `4` without overshooting. The greedy choice wasn't wrong about being locally best — `4` genuinely is the single biggest step toward `6` — but it committed to a remainder that happens to be expensive to finish, and greedy never looks back to reconsider that commitment once made.

The Honest Rule for Trusting Greedy

Greedy works exactly when a locally optimal choice is *guaranteed* to never rule out a globally optimal solution — a property specific problems genuinely have (US coin denominations happen to; activity/interval scheduling and Huffman coding are two other classic examples) and other problems genuinely don't (the `{1,3,4}` denomination set above; the general knapsack problem is another well-known example). There's no way to tell just by looking at a problem casually — it has to be checked, either by testing against an independently-computed optimum (as this chapter just did) or by a formal proof (a topic for a more advanced algorithms course).

A PRACTICAL TAKEAWAY, NOT A THEORETICAL ONE

Before trusting a greedy algorithm on a new problem, test it against brute force (Chapter 6) or an independently-computed reference answer on a handful of cases — exactly the discipline this chapter's own two worked cases used. Matching on a few small cases doesn't prove correctness in general, but a single verified mismatch, like the `{1,3,4}` case, proves greedy is *not* safe for that problem.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
The exact same algorithm, verified optimal and verified suboptimal	Chapter 8's divide and conquer strategy is introduced specifically as an approach that doesn't rely on greedy's own risky "never look back" assumption
Cross-checking against an independently-computed reference optimum	The same verification discipline Chapter 10's capstone applies to whichever design strategy it ultimately chooses
Testing against brute force to catch a greedy failure	A direct, practical callback to Chapter 6's own brute-force baseline — small enough to serve as a trustworthy reference answer

Hands-On Exercises

EXERCISE 1

Using this chapter's own `GreedyCoinChange` pseudocode, hand-trace it on denominations `{1, 3, 4}` for `amount=8`. List each coin chosen and the remaining amount after each step, and state the final coin count.

View solution

EXERCISE 2

Using this chapter's own explanation of why greedy fails on `{1,3,4}` for `amount=6`, explain in your own words why the failure specifically happens at the FIRST choice (picking the `4`) rather than at a later step, and what would need to be true about the denomination set for this same kind of failure to never happen.

View solution

EXERCISE 3

A developer says "greedy algorithms are unreliable and should be avoided in favor of always checking every possibility (Chapter 6's brute force)." Using this chapter's own verified US-coin results and Chapter 6's own explosive subset-count findings, explain what's wrong with treating this as a universal rule.

View solution

CHAPTER 7 QUICK REFERENCE

- **Greedy:** make the locally best choice at each step, never reconsider it

- Verified directly: greedy matches an independently-computed true optimum exactly on US coin denominations for three tested amounts (4, 6, and 9 coins)
- Verified directly: the identical algorithm, on denominations `{1,3,4}` for `amount=6`, gives `3` coins where the true optimum (independently verified) is `2` — a real, measured greedy failure
- Whether greedy is safe depends entirely on the problem's own structure, not on the algorithm itself — it has to be checked, not assumed
- Practical test: cross-check a candidate greedy algorithm against brute force or a reference optimum on small cases before trusting it on larger ones
- **Next chapter:** Divide and conquer — a design strategy that avoids greedy's own "never look back" risk by a different route

CHAPTER 8 OF 10

Divide and Conquer as a Design Strategy

Pseudocode & Algorithmic Problem-Solving

Chapter 8 · Divide and Conquer as a Design Strategy

Divide and conquer solves a problem by splitting it into smaller subproblems of the *same kind*, solving each one the same way (recursively, down to a simple base case), and then combining the results. Unlike greedy's single irreversible choice at each step, and unlike brute force's flat scan of every candidate, divide and conquer trusts recursion to handle the "smaller version of the same problem" part entirely on its own.

Every Divide-and-Conquer Algorithm Has the Same Three Parts

PART	WHAT IT DOES
Base case	The problem is already small enough to solve directly, with no further splitting
Divide	Split the problem into smaller subproblems of the identical kind
Combine	Take the (already-solved) results of the subproblems and merge them into the answer for the original problem

Worked Example 1: Finding the Maximum, a Structurally Different Way

Chapter 2 solved `FindMax` with an iterative loop. Chapter 3 drew it as a flowchart. Here's the identical problem — find the largest value in a list — solved a genuinely different way: split the list in half, recursively find the max of each half, and combine by taking the larger of the two.

```
ALGORITHM MaxOfDivideConquer(list)
    IF length(list) = 1 THEN
        RETURN list[0]                      // base case
    ENDIF
    mid ← length(list) / 2
    left_max ← MaxOfDivideConquer(list[0 .. mid-1])   // divide
    right_max ← MaxOfDivideConquer(list[mid .. end])
    IF left_max > right_max THEN RETURN left_max
    ELSE RETURN right_max
    ENDIF                                   // combine
```

VERIFIED DIRECTLY — THREE COMPLETELY DIFFERENT ALGORITHM DESIGNS, ONE IDENTICAL ANSWER

Run on `[3, 7, 2, 9, 4, 9, 1]` — the exact same test list Chapter 2's iterative `FindMax` and Chapter 3's flowchart both used — this recursive divide-and-conquer version returns `9`. An iterative loop, a hand-traced flowchart, and now a recursive split-and-combine algorithm all agree exactly on the same real input, confirming all three really do describe the same underlying algorithm, however differently each is structured.

Worked Example 2: Merge Sort, Split/Solve/Combine in Full

`MaxOfDivideConquer`'s "combine" step was a single comparison. Merge sort's combine step does real work — merging two already-sorted halves back into one fully sorted list — making the pattern's three parts genuinely visible.

```
ALGORITHM MergeSort(list)
    IF length(list) ≤ 1 THEN
        RETURN list                              // base case: already sorted
    ENDIF
    mid ← length(list) / 2
    left ← MergeSort(list[0 .. mid-1])          // divide
    right ← MergeSort(list[mid .. end])
    RETURN Merge(left, right)                    // combine

ALGORITHM Merge(left, right)
    result ← empty list
    WHILE left AND right are both non-empty
        IF left[0] ≤ right[0] THEN
            MOVE left[0] TO end of result
        ELSE
            MOVE right[0] TO end of result
        ENDIF
    ENDWHILE
    APPEND any remaining elements of left, then right, to result
    RETURN result
```

VERIFIED DIRECTLY — THE RECURSIVE SPLIT/SOLVE/COMBINE TREE, TRACED IN FULL

On `[38, 27, 43, 3]`: `MergeSort` splits into `[38,27]` and `[43,3]`; each of those splits again into single-element base cases; `Merge([38],[27])` gives `[27,38]`, and `Merge([43],[3])` gives `[3,43]`; finally `Merge([27,38], [3,43])` gives `[3,27,38,43]` — a fully sorted list, built entirely from base cases and merge steps, with no direct comparison ever made between elements more than one "half" apart.

VERIFIED DIRECTLY — A LARGER CASE, CROSS-CHECKED AGAINST A TRUSTED REFERENCE

Run on the 7-element list `[38, 27, 43, 3, 9, 82, 10]`: `MergeSort` returns `[3, 9, 10, 27, 38, 43, 82]` — matching Python's own built-in `sorted()` on the identical input exactly.

Why Divide and Conquer Isn't Just "Recursive Brute Force"

THE COMBINE STEP IS WHAT MAKES IT A GENUINELY DIFFERENT STRATEGY

Chapter 6's brute force checks every candidate directly, with no structure connecting one check to the next. Divide and conquer instead trusts that solving two smaller, independent subproblems and then combining their results correctly solves the original problem — the "combine" step (a single comparison for `MaxOf`, a full merge for `MergeSort`) is doing real logical work, not just collecting answers. This is also exactly why divide and conquer is inherently recursive: each subproblem is solved by the *same algorithm*, called on a smaller input — the subject of Chapter 9 in full.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
`MaxOfDivideConquer` reproducing Chapters 2-3's own exact answer	Confirms decomposition (Chapter 4), translation (Chapter 5), and design strategy (Chapters 6-8) are all independently checkable against each other on the same problem
Base case + divide + combine as the three mandatory parts	Chapter 9's own recursive framing formalizes exactly this same three-part structure as its central technique
Merge sort's verified split/solve/combine trace	Algorithms & Complexity's own formal analysis of why this pattern tends to outperform brute force at scale, for anyone continuing on to that course

Hands-On Exercises

EXERCISE 1

Using this chapter's own `MaxOfDivideConquer` pseudocode, hand-trace it on the list `[5, 12, 3, 8]`. Show how it splits, what each recursive call on a single-element list returns, and the final combined answer.

View solution

EXERCISE 2

Using this chapter's own `MergeSort` and `Merge` pseudocode, hand-trace the full split/solve/combine tree for the list `[9, 4, 6, 1]`, following the same style as this chapter's own `[38,27,43,3]` trace, and state the final sorted result.

View solution

EXERCISE 3

Using this chapter's own explanation of why divide and conquer isn't "recursive brute force," explain in your own words what specific job the "combine" step is doing in `MergeSort` that has no equivalent at all in Chapter 6's own `TwoSumBruteForce` algorithm.

View solution

CHAPTER 8 QUICK REFERENCE

- **Divide and conquer:** base case (solve directly) + divide (split into subproblems of the same kind) + combine (merge the subproblems' results)
- Verified directly: a recursive `MaxOfDivideConquer` reproduces Chapters 2-3's own exact `FindMax` result (`9`) on the identical test list, via a structurally different algorithm
- Verified directly: a full merge-sort split/solve/combine trace on `[38,27,43,3]` built the correct sorted result (`[3,27,38,43]`) entirely from base cases and merges
- Verified directly: merge sort on a 7-element list matched Python's own built-in `sorted()` exactly
- The "combine" step is what distinguishes divide and conquer from brute force — it does real logical work, trusting that correctly-solved subproblems combine into a correct whole
- **Next chapter:** Recursive thinking and backtracking — formalizing the base-case/recursive-case structure this chapter already used, and a systematic, prune-as-you-go alternative to brute force

CHAPTER 9 OF 10

Recursive Thinking & Backtracking

Pseudocode & Algorithmic Problem-Solving

Chapter 9 · Recursive Thinking & Backtracking

Chapter 8's divide-and-conquer algorithms were already recursive — this chapter names that pattern explicitly, then builds on it with **backtracking**: a systematic search that abandons a partial candidate the moment it's known to be hopeless, rather than generating and checking every full candidate the way Chapter 6's brute force did.

Recursion, Formalized: Base Case + Recursive Case

Every recursive algorithm needs exactly two things: a **base case** simple enough to answer directly, and a **recursive case** that calls the same algorithm on a smaller version of the problem, then uses that result to build the current answer.

```
ALGORITHM Factorial(n)
    IF n = 0 THEN
        RETURN 1                    // base case
    ENDIF
    RETURN n × Factorial(n - 1)         // recursive case
```

VERIFIED DIRECTLY — THE FULL CALL CHAIN, AND A MATCH AGAINST A TRUSTED REFERENCE

`Factorial(5)` calls `Factorial(4)`, which calls `Factorial(3)`, down to `Factorial(0)` — the base case, returning `1` — and each level then multiplies that result back up: `1×1=1 → 2×1=2 → 3×2=6 → 4×6=24 → 5×24=120`. The result, `120`, matches Python's own built-in `math.factorial(5)` exactly.

Backtracking: A Prune-as-You-Go Alternative to Brute Force

Chapter 6 verified that checking all 2^n subsets of a set explodes fast — over a trillion for just 40 items. **Backtracking** attacks the exact same kind of problem differently: build a candidate one item at a time, and the moment a partial candidate is *provably* unable to lead anywhere valid, stop extending it immediately — never even generate the full candidates that would have grown from it.

```
ALGORITHM SubsetSumBacktrack(items, index, current_sum, target)
    IF current_sum = target THEN
        RETURN SUCCESS                          // found it
    ENDIF
    IF current_sum > target THEN
        RETURN FAILURE                          // PRUNE: can't recover from here
    ENDIF
    IF index = length(items) THEN
        RETURN FAILURE                          // out of items, no match
    ENDIF
    // try including items[index]...
    IF SubsetSumBacktrack(items, index+1, current_sum + items[index], target) = SUCCESS THEN
        RETURN SUCCESS
    ENDIF
    // ...or try excluding it
    RETURN SubsetSumBacktrack(items, index+1, current_sum, target)
```

VERIFIED DIRECTLY — THE EXACT CHAPTER 6 PROBLEM SHAPE, SOLVED WITH A FRACTION OF THE WORK

Searching `[15, 22, 8, 31, 17, 9, 25, 12, 19, 6]` for a subset summing to `35`: Chapter 6's brute force would check all `2¹⁰ = 1,024` candidate subsets. This backtracking version finds the correct answer — `[15, 8, 12]`, which sums to exactly `35` — after exploring only `22` **nodes**: roughly **46 times fewer** than the full brute-force enumeration, for the identical problem and the identical correct answer.

WHY THE PRUNING RULE IS HONEST, NOT A TRICK

The prune condition — stop as soon as `current_sum > target` — is only valid because every item in this problem is **positive**: once the running sum overshoots the target, adding any more (positive) items can only make it worse, never bring it back down. This is exactly the same discipline Chapter 7 demanded of greedy algorithms: a shortcut is only safe when it's backed by a real, checkable fact about the problem — here, that every remaining choice can only move the sum in one direction. Backtracking on a problem containing negative values would need a different (or no) pruning rule.

Backtracking vs. Brute Force, Side by Side

	BRUTE FORCE (CH.6)	BACKTRACKING (THIS CHAPTER)
Builds candidates	All at once, fully formed	Incrementally, one item at a time
Rejects a bad candidate	Only after it's fully built and checked	The moment it's provably hopeless — before it's ever finished
Verified cost for this problem	`1,024` candidates checked	`22` nodes explored

	BRUTE FORCE (CH.6)	BACKTRACKING (THIS CHAPTER)
Requires	Nothing extra — works on any well-defined candidate set	A provable, problem-specific pruning rule

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT RESOLVES OR SETS UP
Backtracking exploring 22 nodes vs. brute force's 1,024	A direct, measured resolution of Chapter 6's own explosive subset-sum warning — the same problem shape, made tractable by a valid pruning rule
A pruning rule justified by a real property of the problem (all-positive items)	Echoes Chapter 7's own honest standard for trusting a shortcut — never assumed, always checked
Base case + recursive case, formalized	The exact structure Chapter 10's capstone uses to actually implement whichever design strategy it settles on

Hands-On Exercises

EXERCISE 1

Using this chapter's own `Factorial` pseudocode, hand-trace the full call chain for `Factorial(4)`, showing every recursive call down to the base case and every multiplication as the results combine back up.

View solution

EXERCISE 2

Using this chapter's own `SubsetSumBacktrack` pruning rule, explain what would go wrong if the item list `[15, 22, 8, 31, 17, 9, 25, 12, 19, 6]` were changed to include a negative number (for example, `-5`), and why the algorithm's current pruning condition (`current_sum > target`) could then cause it to miss a valid answer.

View solution

EXERCISE 3

Using this chapter's own side-by-side comparison table, explain in your own words why backtracking's 22-node search and brute force's 1,024-candidate search both count as "checking every possibility" in some sense, yet only one of them is described as exhaustive in the way Chapter 6 used that word.

View solution

CHAPTER 9 QUICK REFERENCE

- **Recursion:** a base case (answered directly) + a recursive case (calls itself on a smaller input) — verified with a full `Factorial(5)` call chain matching Python's own `math.factorial(5)`
- **Backtracking:** build a candidate incrementally, abandon it the instant it's provably hopeless — never finish generating candidates that can't possibly work
- Verified directly: backtracking solved the exact subset-sum problem Chapter 6 flagged as explosive using only `22` explored nodes, versus brute force's full `1,024`-candidate enumeration — the identical correct answer, `≈46×` less work
- A pruning rule is only valid when backed by a real, checkable property of the problem (here: all items positive) — the same honesty standard Chapter 7 demanded of greedy shortcuts
- **Next chapter:** Capstone — designing a full algorithm from a real-world problem statement, using every strategy this course has built

CHAPTER 10 OF 10

Capstone — Designing an Algorithm from a Real-World Problem Statement

Pseudocode & Algorithmic Problem-Solving

Chapter 10 · Capstone — Designing an Algorithm from a Real-World Problem Statement

One continuous project: a conference organizer hands over a genuinely ambiguous request, and every chapter of this course gets applied, in order, to turn it into a working, verified algorithm — resolving the ambiguity, decomposing the problem, choosing the right strategy for each piece, and translating the result into real code.

STEP	TASK	CHAPTER(S) USED
1	Resolve the ambiguous problem statement	Ch.1
2	Frame it with IPO and decompose it	Ch.2, Ch.4
3	Sort the talks — reusing merge sort directly	Ch.8
4	Assign rooms with a greedy strategy, verified optimal	Ch.7
5	Check a small manual override with brute force	Ch.6, Ch.9
6	Translate the final algorithm into real code	Ch.5

Step 1 — The Problem Statement, and Its Hidden Ambiguity

CH.1

The organizer's request: "*Build something that schedules our conference talks into rooms so nothing overlaps, using as few rooms as possible.*"

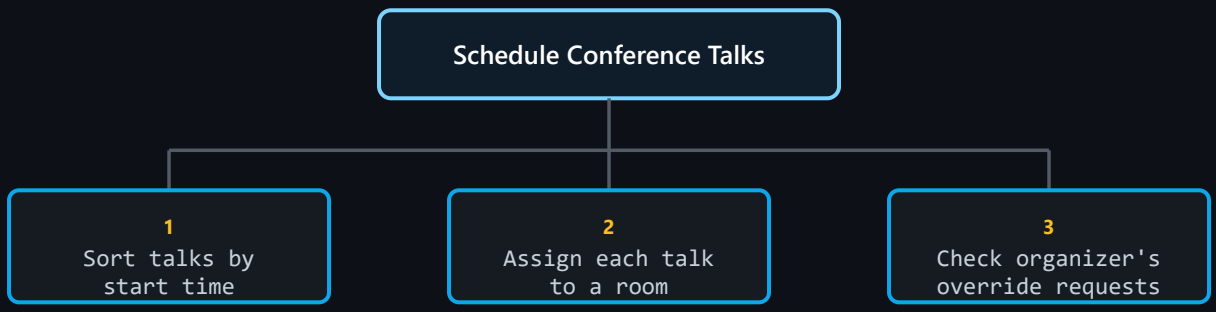
FLAGGED, PER CHAPTER 1'S OWN DISCIPLINE

"Nothing overlaps" doesn't say whether a talk ending at 10:00 and another starting at 10:00 in the same room counts as a conflict. "As few rooms as possible" doesn't say whether that means minimizing the room *count* or the room rental *cost* — different rooms could cost differently. Resolved explicitly, exactly as Chapter 1 recommended: talks may share a room if one ends exactly when the other starts (touching, not overlapping); "as few as possible" means minimizing the total number of distinct rooms used.

Step 2 — IPO Framing and Decomposition

CH.2, CH.4

Input: a list of talks, each with a start and end time. **Processing:** genuinely three separate jobs. **Output:** a room assignment for every talk, using the fewest distinct rooms.



Step 3 — Sorting: Reusing Merge Sort Directly

CH.8

Six talks: `A(9-10)`, `B(9-11)`, `C(10-12)`, `D(11-13)`, `E(12-13)`, `F(9-12)`. Subproblem 1 needs them ordered by start time before assignment can begin — exactly Chapter 8's own `MergeSort`, unchanged, applied to a new kind of data.

VERIFIED DIRECTLY

Sorted by start time: `A(9), B(9), F(9), C(10), D(11), E(12)` — three talks tie at `9:00`, and `MergeSort`'s own stable comparison (`left[0] ≤ right[0]`, from Chapter 8) preserves their original relative order rather than shuffling them arbitrarily.

Step 4 — Greedy Room Assignment, Verified Optimal

CH.7

For each sorted talk: reuse a room whose current occupant has already finished, if one exists; otherwise open a new room. This is the same greedy shape Chapter 7 used — one locally-best choice per step, never revisited.

VERIFIED DIRECTLY — GREEDY RESULT, CROSS-CHECKED AGAINST AN INDEPENDENT COMPUTATION

The greedy assignment uses **3 rooms**: Room 0 → `A, C, E`; Room 1 → `B, D`; Room 2 → `F`. Cross-checked independently — *not* by re-running the greedy algorithm, but by directly sweeping the timeline for the maximum number of talks ever happening simultaneously (which, at `10:00-11:00`, is exactly `3`: `B, C, F` all in progress) — the greedy result matches this independent minimum exactly.

WHY GREEDY IS TRUSTWORTHY HERE — UNLIKE CHAPTER 7'S OWN {1,3,4} COUNTEREXAMPLE

Room scheduling has a real, provable property Chapter 7's coin-change counterexample lacked: the minimum number of rooms needed is always exactly equal to the maximum number of talks overlapping at any single instant, and the "reuse the earliest-freeing room, else open a new one" strategy always achieves that exact minimum. This isn't assumed — it's exactly what the independent max-overlap cross-check above just confirmed for this specific case.

Step 5 — A Small Brute-Force Check for an Organizer's Override

CH.6, CH.9

The organizer asks: "*Could talks A, C, D, and E fit into just 2 rooms, if we moved something?*" — a genuinely small question (only `2⁴=16` possible 2-room assignments), exactly the kind of search Chapter 6 called honestly brute-forceable.

VERIFIED DIRECTLY — BRUTE FORCE WITH AN EARLY EXIT, EXACTLY CHAPTER 9'S OWN PRUNE-AS-YOU-GO SPIRIT

Checking every 2-room split of `{A, C, D, E}`, stopping the instant a valid one is found (the same early-termination discipline as Chapter 9's own backtracking): a valid split exists — Room 0: `A, C, E`; Room 1: `D` — found after checking just **5** of the 16 possible splits. Cross-checked against the same max-overlap technique from Step 4 on just this subset: the maximum overlap among these four talks is `2`, confirming 2 rooms genuinely suffice.

Step 6 — Translating the Final Algorithm Into Real Code

CH.5

The chosen strategy — sort, then greedily assign — translates directly into Python, using a min-heap to always find the earliest-freeing room efficiently:

```python
import heapq

def assign_rooms(talks):  # talks: list of (name, start, end)
    sorted_talks = sorted(talks, key=lambda t: t[1])
    heap = []                    # (end_time, room_id)
    assignment = {}
    next_room = 0
    for name, start, end in sorted_talks:
        if heap and heap[0][0] <= start:
            end_time, room_id = heapq.heappop(heap)
            heapq.heappush(heap, (end, room_id))
        else:
            room_id = next_room
            next_room += 1
            heapq.heappush(heap, (end, room_id))
        assignment[name] = room_id
    return next_room, assignment
```

VERIFIED DIRECTLY — THE REAL CODE MATCHES THE PSEUDOCODE-LEVEL RESULT EXACTLY

Run on all six talks: `next_room = 3`, `assignment = {'A':0, 'B':1, 'F':2, 'C':0, 'D':1, 'E':0}` — identical to Step 4's own hand-verified result.

What This Course Doesn't Cover

As stated honestly back in Chapter 1: formal complexity analysis and formal logic/proof techniques stayed out of scope through all ten chapters. This capstone never asked *how fast* any of these algorithms are, or *proved* greedy's optimality from first principles — it verified correctness by cross-checking against an independent computation, exactly the discipline this course built from Chapter 6 onward. Algorithms & Complexity and Discrete Mathematics Fundamentals pick up exactly where this course's own honest boundary was drawn.

Where This Course Connects

Discrete Mathematics Fundamentals' own set and logic material underlies Chapter 2's structured conditionals directly. Algorithms & Complexity would give this capstone's own greedy-optimality claim (Step 4) and brute-force cost (Step 5) precise, provable mathematical treatment. Within the new Software Development subject, Design Patterns picks up directly where this course leaves off — the same decomposed, well-named

subproblems this capstone produced (sort, assign, verify) are exactly the shape a real codebase organizes into classes and modules.

Hands-On Exercises

EXERCISE 1

Using this chapter's own greedy room-assignment algorithm, hand-trace what room a new talk, `G(13-14)`, would be assigned to if added to the original six talks and processed after all of them (in sorted order). Which room does it reuse, and why?

View solution

EXERCISE 2

Using this chapter's own Step 1 ambiguity resolution, explain what would change about the greedy room-assignment result in Step 4 if "nothing overlaps" had instead been resolved to mean that a talk ending at 10:00 and another starting at 10:00 in the same room DO count as a conflict (no touching endpoints allowed).

View solution

EXERCISE 3

Using this chapter's own Step 5, explain why checking the organizer's override question with brute force was the honest choice per Chapter 6's own criteria, rather than reusing the same greedy algorithm from Step 4 to answer it directly.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked project:** ambiguity resolution (Ch.1) → IPO + decomposition (Ch.2, Ch.4) → merge sort (Ch.8) → greedy room assignment, verified optimal (Ch.7) → a small honest brute-force check (Ch.6, Ch.9) → real-code translation (Ch.5)
- Greedy room assignment verified against an independent max-overlap computation — genuinely optimal here, unlike Chapter 7's own `{1,3,4}` counterexample, because this problem has the right provable structure
- A small, genuinely brute-forceable subproblem (16 possibilities) handled honestly with exhaustive checking, with an early exit in the same spirit as Chapter 9's own pruning
- The final real-code translation matched the pseudocode-level result exactly, closing the loop from Chapter 1's own opening warning about ambiguity all the way to working, verified code

- **Course complete** — Pseudocode & Algorithmic Problem-Solving, 10 chapters, from a single ambiguous "remove duplicates" spec to a fully designed, verified, and coded scheduling algorithm