Algorithms & Complexity

A Complete 10-Chapter Maths for Programmers Course

Topics covered:

Big-O, Big-Omega & Big-Theta · loop and recursion analysis

The Master Theorem · searching & sorting · space & amortized analysis

Exponential growth & a taste of P vs. NP

Capstone: comparing three real algorithms for the same problem

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 Algorithmic Complexity Matters for Programmers

Algorithms & Complexity

Chapter 1 · Why Algorithmic Complexity Matters for Programmers

Two pieces of code can produce the exact same correct output and still be worlds apart in practice — one finishes instantly no matter how much data you throw at it, the other quietly grinds to a halt the moment real-world scale shows up. Complexity analysis is the math for predicting which one you've written, before it becomes a production incident.

What Complexity Analysis Actually Measures

Not raw seconds — that depends on hardware, language, and a dozen other details that change from machine to machine. Complexity analysis measures something more durable: how the **number of operations** (or the amount of memory) an algorithm needs *grows* as the input size grows. It's a statement about shape, not speed.

A STRIKING REAL COMPARISON

Searching a sorted array of **1,000,000** elements: a linear scan needs up to **1,000,000** comparisons in the worst case. Binary search needs only $\lceil \log_2(1{,}000{,}000) \rceil = 20$. Same task, same correct answer — a 50,000× difference in worst-case work, purely from choosing a different algorithm.

Five Concrete Connections to Code Already On This Site

COMPLEXITY TOPIC	WHERE IT ACTUALLY SHOWS UP
Big-O & growth rates (Ch.2)	Why a dictionary/set lookup is effectively instant (O(1)) while searching a plain list is not (O(n)) — a real, measurable performance difference in everyday code
Analyzing loops (Ch.3–4)	Spotting that an innocent-looking nested loop has quietly made a function $O(n^2)$ — the single most common accidental performance bug
Recursion & recurrences (Ch.5–6)	Understanding why one recursive solution explodes exponentially while a structurally similar one (like merge sort) stays efficient
Searching & sorting (Ch.7)	Knowing when sorting data first, then searching repeatedly, beats scanning linearly every single time

COMPLEXITY TOPIC	WHERE IT ACTUALLY SHOWS UP
Real relevance	Technical Support's own `perfdiag1` asks "why is this slow" from a diagnostic, after-the-fact angle; this course is the proactive, code-level version of that exact same question

What This Course Won't Cover

A few genuinely related areas are deliberately left out, to keep this course focused specifically on the *math* of measuring efficiency:

- **A full catalog of algorithms** — this course teaches how to *analyze* any algorithm's complexity, not a comprehensive tour of every sorting/searching algorithm's own implementation details; a general Programming-subject course would be the place for that breadth
- **Graph algorithms specifically** — pathfinding, traversal, and network analysis are reserved for this subject's own future Graph Theory course
- **Deep computational complexity theory** — Chapter 9 gives P vs. NP a light, honest conceptual treatment, not a formal complexity-theory course

> **WHY DRAW THE LINE HERE INSTEAD OF COVERING EVERYTHING AT ONCE**
> Each of those areas is substantial enough to deserve its own real depth. This course stays tightly scoped to the actual mechanics of measuring and reasoning about efficiency — the direct foundation those broader topics would each build on.

Watch These Numbers Diverge

The whole point of complexity classes is how differently they scale. A few common classes, at increasing input sizes:

N	$\log_2 N$	N	$N \log_2 N$	N^2	2^n
5	2.3	5	11.6	25	32
10	3.3	10	33.2	100	1,024
20	4.3	20	86.4	400	1,048,576

By `n = 20`, the $O(n^2)$ column has grown 20× from its own `n=5` value — and the $O(2^n)$ column has grown over 32,000× from its own `n=5` value. Same starting point, wildly different destinies. Chapter 9 pushes this comparison even further.

Where This Course Is Headed

CHAPTER	TOPIC
2	Big-O Notation: Formal Definition & Growth Rates
3	Analyzing Loops: From Code to Big-O
4	Big-Omega & Big-Theta: Best, Worst & Average Case
5	Recursive Algorithms & Recurrence Relations
6	Solving Recurrences: Substitution & the Master Theorem
7	Common Complexity Classes in Practice: Searching & Sorting
8	Space Complexity & Amortized Analysis
9	Beyond Polynomial Time: Exponential Growth & a Taste of P vs. NP
10	Capstone — Analyzing and Comparing Real Algorithms

THIS COURSE'S THROUGHLINE

Every chapter answers a version of the same question: as the input gets bigger, how does the work required actually grow — and can that growth be predicted from the code itself, before ever running it? That's the skill that separates "this works on my test data" from "this still works when a customer uploads ten million rows."

Hands-On Exercises

EXERCISE 1

A sorted array has 100,000 elements. Compute the worst-case number of comparisons for a linear scan, and the worst-case number of comparisons for binary search ($\lceil \log_2(n) \rceil$). Express the ratio between the two as a single number.

View solution

EXERCISE 2

A colleague claims "complexity analysis doesn't matter — modern computers are fast enough that it's never worth thinking about." Using this chapter's own growth-rate table, explain why this claim breaks down specifically for $O(n^2)$ and $O(2^n)$ algorithms as input size grows, even on very fast hardware.

View solution

EXERCISE 3

For each of the following, name which topic from this chapter's own five-connections table it most directly maps to, and explain the connection in one or two sentences: (a) a function that checks whether a username already exists by scanning a Python list of all existing usernames; (b) a function with two nested loops comparing every pair of items in a list; (c) a function that repeatedly halves a sorted list to find a target value.

View solution

CHAPTER 1 QUICK REFERENCE

- Complexity analysis measures how the **number of operations** grows with input size — a statement about shape, not raw seconds
- Linear search vs. binary search over 1,000,000 elements: 1,000,000 comparisons vs. 20 — a 50,000× difference from algorithm choice alone
- Five direct connections: dict/set O(1) lookups, spotting accidental $O(n^2)$ nested loops, why some recursion explodes and some doesn't, sort-then-search vs. repeated linear search, and Technical Support's own diagnostic mindset applied proactively
- Deliberately out of scope here: a full algorithms/data-structures catalog, graph algorithms (reserved for a future Graph Theory course), and deep complexity theory
- Different complexity classes diverge dramatically even at small input sizes — $O(2^n)$ overtakes $O(n^2)$ astonishingly fast
- **Next chapter:** Big-O notation — formal definition and growth rates

CHAPTER 2 OF 10

Big-O Notation: Formal Definition & Growth Rates

Algorithms & Complexity

Chapter 2 · Big-O Notation: Formal Definition & Growth Rates

Chapter 1 used Big-O informally — "$O(n^2)$" as a label for "the work grows like the square of the input." This chapter makes that label precise, with a real formal definition, and builds the full ranked ladder of complexity classes it describes.

The Formal Definition

BIG-O, FORMALLY

$f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$.

In plain terms: $f(n)$ is O(g(n)) if, past some point (n_0), $f(n)$ never grows faster than a constant multiple of $g(n)$. Big-O describes an **upper bound** on growth — "grows no faster than" — ignoring exactly how much faster or slower below that bound the real behavior is, and ignoring everything that happens before n_0.

Verifying the Definition With Real Numbers

Claim: $f(n) = 3n^2 + 5n + 100$ is $O(n^2)$. Trying $c = 4$:

N	F(N) = 3N²+5N+100	4N²	F(N) ≤ 4N²?
11	518	484	No
12	592	576	No
13	672	676	**Yes**
14	758	784	Yes

With $c = 4$ and $n_0 = 13$, the definition holds for every $n \ge 13$ — confirmed exactly at the boundary. $f(n) = 3n^2 + 5n + 100$ is genuinely, verifiably $O(n^2)$, not just "probably" or "roughly."

The Dominant-Term Rule

In practice, nobody hunts for `c` and `n₀` by hand every time. The shortcut: drop every term except the fastest-growing one, and drop its constant coefficient too.

DOMINANT-TERM SIMPLIFICATION

`3n² + 5n + 100` simplifies to `O(n²)` — the `n²` term dominates as `n` grows; the `5n` and `100` terms, and even the leading `3`, become irrelevant to the growth *shape*.

Why this is valid — watch the ratio of the full expression to just `n²` as `n` grows:

N	F(N) = 3N²+5N+100	N²	F(N) / N²
10	450	100	4.50
100	30,600	10,000	3.06
1,000	3,005,100	1,000,000	3.005

THE RATIO CONVERGES TO A CONSTANT, NOT ZERO OR INFINITY

As `n` grows, that ratio settles toward exactly `3` — the leading coefficient. It doesn't shrink to nothing (which would mean `n²` overstates the growth) or blow up (which would mean `n²` understates it) — it converges to a fixed multiple, which is precisely what "constant `c`" means in the formal definition above. The dominant-term shortcut isn't a hand-wave; it's the formal definition, worked out in advance for the general case.

The Ranked Ladder of Common Complexity Classes

CLASS	NAME	REAL EXAMPLE
O(1)	Constant	Array index access, hash table lookup
O(log n)	Logarithmic	Binary search (Chapter 1's own example)
O(n)	Linear	A single pass over a list, linear search
O(n log n)	Linearithmic	Efficient sorting — merge sort, average-case quicksort
O(n²)	Quadratic	Nested loops, bubble sort
O(n³)	Cubic	Triple-nested loops, naive matrix multiplication
O(2ⁿ)	Exponential	Naive recursive Fibonacci, generating every subset
O(n!)	Factorial	Generating every permutation, brute-force traveling salesman

CONVENTION: ALWAYS STATE THE TIGHTEST VALID BOUND

Technically, an O(1) algorithm is also, correctly, O(n) — a constant is never larger than a constant multiple of `n` past some point, so the formal definition holds. But stating "O(n)" for a genuinely constant-time algorithm would be true yet uselessly loose. By strong convention, Big-O is always reported as the *smallest* (tightest) class that still validly bounds the function — Chapter 4's Big-Theta gives this convention a fully formal footing.

Big-O Verification in Code

```
def f(n):
    return 3*n**2 + 5*n + 100

def is_O_n_squared(f, c, n0, test_range=range(1, 100)):
    return all(f(n) <= c * n**2 for n in test_range if n >= n0)

print(is_O_n_squared(f, c=4, n0=13))  # True — matches the worked verification above
print(is_O_n_squared(f, c=4, n0=1))   # False — n0=13 is genuinely required, not just convenient
```

Hands-On Exercises

EXERCISE 1

Using this chapter's own dominant-term rule, simplify `f(n) = 7n³ + 2n² + 50` to its Big-O class. State which terms were dropped and why.

View solution

EXERCISE 2

Verify `f(n) = 2n² + 3n + 10` is `O(n²)` using `c = 3`. Find the smallest integer `n₀` for which `f(n) ≤ 3n²` holds for all `n ≥ n₀`, showing the boundary value where it first becomes true.

View solution

EXERCISE 3

Rank the following complexity classes from slowest-growing to fastest-growing: `O(n²)`, `O(log n)`, `O(1)`, `O(2ⁿ)`, `O(n log n)`, `O(n)`. Briefly justify the placement of `O(n log n)` relative to its two neighbors in your ranking.

View solution

CHAPTER 2 QUICK REFERENCE

- **Formal definition:** $f(n)=O(g(n))$ if $f(n) \le c \cdot g(n)$ for some constants c, n_0 and all $n \ge n_0$
- **Dominant-term rule:** keep only the fastest-growing term, drop its coefficient — valid because the ratio to that term converges to a constant, never zero or infinity
- **Ranked classes:** $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$
- By convention, always report the *tightest* valid Big-O class, even though looser bounds are technically also true
- **Next chapter:** Analyzing loops — from code to Big-O

CHAPTER 3 OF 10

Analyzing Loops: From Code to Big-O

Algorithms & Complexity

Chapter 3 · Analyzing Loops: From Code to Big-O

Chapter 2 built the math; this chapter is where it gets applied directly to real code. The core technique is almost mechanical once you know the rules: **count how many times each line of code runs, as a function of `n`, then simplify with Chapter 2's own dominant-term rule.**

A Single Loop — O(n)

```
total = 0
for i in range(n):
    total += i
```

The loop body runs exactly `n` times, each iteration doing a fixed, constant amount of work. Total: `O(n)`.

Sequential Loops — The Sum Rule

```
for i in range(n):
    print(i)

for j in range(n):
    print(j)
```

SUM RULE

Two separate (not nested) blocks of code each running `n` times: total work is `n + n = 2n`, which the dominant-term rule collapses straight to `O(n)`. Sequential blocks **add**; the slowest-growing block among them never changes the overall class, only its irrelevant constant.

Nested Loops (Independent Bounds) — The Product Rule

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

PRODUCT RULE

The inner loop runs `n` times for *each* of the outer loop's `n` iterations: `n × n = n²` total. Nested blocks **multiply**. This is exactly Chapter 1's own "innocent-looking nested loop" — two ordinary `for` loops, quietly $O(n^2)$.

Triangular Loops (Dependent Bounds) — Still $O(n^2)$

```
for i in range(n):
    for j in range(i):
        print(i, j)
```

Here the inner bound depends on the outer loop variable — the total operation count is a sum: `0 + 1 + 2 + ... + (n−1)`. This is exactly the summation Discrete Mathematics Fundamentals Chapter 8 proved a closed form for: $\Sigma_{i=0}^{n-1} i = n(n-1)/2$.

VERIFIED WITH A REAL N

At `n = 6`: direct sum `0+1+2+3+4+5 = 15`; formula `6×5/2 = 15` — exact match. And `n(n−1)/2` simplifies (Chapter 2's own dominant-term rule) to `O(n²)` — the same class as the fully independent nested loop above, just with roughly half the actual operations.

Gotcha 1: A Constant Inner Bound Is Not $O(n^2)$

```
for i in range(n):
    for j in range(10):   # fixed, always 10 — not related to n
        print(i, j)
```

NESTING ALONE DOESN'T MEAN $O(N^2)$

Total operations: `n × 10 = 10n`. Since `10` is a fixed constant, entirely unrelated to `n`, this simplifies to `O(n)`, not `O(n²)` — despite the visible nesting. The product rule only produces a genuinely higher complexity class when *both* bounds actually grow with `n`.

Gotcha 2: A Multiplicative Loop Variable Gives O(log n)

```
i = 1
while i < n:
    i *= 2   # doubling, not incrementing by 1
```

Because `i` *doubles* each iteration rather than simply incrementing, the loop runs far fewer than `n` times. For `n = 1,000,000`: this loop runs exactly **20** times before `i` reaches or exceeds `n` — matching `log₂(1,000,000) ≈ 19.93`, and matching Chapter 1's own binary search figure exactly. Whenever a loop variable grows *multiplicatively*, the complexity is `O(log n)`, not `O(n)`.

Loop Analysis in Code

```
def count_triangular_operations(n):
    count = 0
    for i in range(n):
        for j in range(i):
            count += 1
    return count

n = 6
print(count_triangular_operations(n))  # 15
print(n * (n - 1) // 2)             # 15 — matches the closed-form formula exactly

def count_doubling_iterations(n):
    i, count = 1, 0
    while i < n:
        i *= 2
        count += 1
    return count

print(count_doubling_iterations(1_000_000))  # 20
```

Hands-On Exercises

EXERCISE 1

A function runs one loop over `n` items, then (not nested — sequentially afterward) a second loop over a separate, fixed-size list of exactly 50 items, then a third loop over the same `n` items again. Using this chapter's own sum rule, determine the overall Big-O class, and explain why the 50-item loop doesn't change the final answer.

View solution

EXERCISE 2

For a triangular loop identical in structure to this chapter's own example, compute the exact total operation count at `n = 10` both by direct summation and by the closed-form formula `n(n-1)/2`, confirming they match. State the resulting Big-O class.

View solution

EXERCISE 3

A loop starts with `i = 1` and triples `i` (`i *= 3`) each iteration until `i ≥ n`. State the Big-O class of this loop, and compute the exact number of iterations it takes to reach or exceed `n = 1,000,000`.

📄 View solution

CHAPTER 3 QUICK REFERENCE

- **Method:** count operations per line as a function of n, sum them up, simplify with Chapter 2's dominant-term rule
- **Sum rule:** sequential (non-nested) blocks add their complexities — the slowest-growing one wins
- **Product rule:** nested blocks multiply their complexities — `n × n = n²` for independent bounds
- **Triangular loops** (dependent inner bound) use the summation formula `Σi = n(n−1)/2` — still $O(n^2)$, just with about half the operations
- **Gotcha 1:** a nested loop with a constant (non-n-dependent) bound stays O(n), not $O(n^2)$
- **Gotcha 2:** a loop variable that grows multiplicatively (doubling, tripling) gives O(log n), not O(n)
- **Next chapter:** Big-Omega and Big-Theta — best, worst, and average case

CHAPTER 4 OF 10

Big-Omega & Big-Theta: Best, Worst & Average Case

Algorithms & Complexity

Chapter 4 · Big-Omega & Big-Theta: Best, Worst & Average Case

Chapter 2 gave Big-O an upper bound and a promise: "Chapter 4's Big-Theta gives this convention a fully formal footing." Here it is — the other two asymptotic bounds, and a second, genuinely different axis this chapter untangles from them: best, worst, and average *case*.

Big-Omega — The Lower Bound

BIG-OMEGA, FORMALLY

$f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Where Big-O says "grows no *faster* than," Big-Omega says "grows no *slower* than" — a floor under the growth, instead of a ceiling.

Big-Theta — The Tight Bound

BIG-THETA, FORMALLY

$f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$ — both bounds hold at once, pinning the growth rate down exactly.

This is the formal version of Chapter 2's own convention — "always report the tightest valid bound" really means: whenever possible, report the **Θ** class, since it's the one that's both an upper *and* lower bound simultaneously.

A Fully Worked Θ Proof

Reusing Chapter 2's own function, $f(n) = 3n^2 + 5n + 100$, already shown to be $O(n^2)$ with $c=4,\ n_0=13$:

BOUND	CONSTANTS	WHY IT HOLDS
$O(n^2)$	c=4, n_0=13	Verified in Chapter 2 — $f(n) \leq 4n^2$ for all $n \geq 13$

BOUND	CONSTANTS	WHY IT HOLDS
$\Omega(n^2)$	$c=3, n_0=1$	`3n² + 5n + 100 ≥ 3n²` since `5n + 100` is always positive — trivially true for every `n ≥ 1`

Both bounds hold with real, verified constants — so `f(n) = Θ(n²)`, not just informally "roughly n^2." Interestingly, the lower bound here was the easier one to prove.

The Other Axis: Best, Worst & Average Case

TWO GENUINELY DIFFERENT QUESTIONS, EASILY CONFLATED

O / Ω / Θ describe bounds on growth *for a given scenario*. **Best/worst/average case** describe *which scenario* — which specific input is being analyzed. They're orthogonal: an algorithm can have a Θ(1) best case and a completely separate Θ(n) worst case, both perfectly valid, tight bounds, just for different inputs.

Worked Example: Linear Search's Three Cases

CASE	SCENARIO	COMPLEXITY
Best	Target is the very first element checked	Θ(1)
Worst	Target is the last element, or absent entirely	Θ(n)
Average	Target equally likely at any of the n positions	Θ(n)

For the average case, if the target sits at position `k` (1-indexed) with equal probability `1/n` for each position, the expected number of comparisons is `(1+2+...+n)/n = (n+1)/2` — verified directly at `n=5`: `(1+2+3+4+5)/5 = 3`, matching `(5+1)/2 = 3` exactly.

N/2 IS STILL Θ(N) — CONSTANTS DON'T SURVIVE BIG-NOTATION

`(n+1)/2` looks like "half the work" of the worst case, and in a literal sense it is — but Chapter 2's own dominant-term rule discards constant factors entirely. Both the worst case and the average case land in the exact same `Θ(n)` class; the factor-of-2 difference is real and worth knowing, but it's invisible to Big-notation by design.

Why Worst Case Matters Even When It's Rare

A SECURITY-ADJACENT REAL CONSEQUENCE

A hash table's *average* case lookup is famously Θ(1) — but its *worst* case, when many keys collide into the same bucket, degrades to Θ(n). This isn't just theoretical: an attacker who can predict or influence how a hash function distributes keys can deliberately craft input that forces every lookup into its worst case, turning a service's normally

fast hash-table operations into a genuine denial-of-service vector. Average-case performance describes typical behavior; worst-case bounds describe what an adversary — or just unlucky data — can actually force.

Verifying Θ in Code

```
def f(n):
    return 3*n**2 + 5*n + 100

def is_Theta_n_squared(f, c_lower, c_upper, n0, test_range=range(1, 100)):
    return all(
        c_lower * n**2 <= f(n) <= c_upper * n**2
        for n in test_range if n >= n0
    )

print(is_Theta_n_squared(f, c_lower=3, c_upper=4, n0=13))  # True

def average_case_linear_search(n):
    return sum(range(1, n + 1)) / n

print(average_case_linear_search(5))  # 3.0 — matches (n+1)/2
```

Hands-On Exercises

EXERCISE 1

Inserting a new element into a sorted array (shifting elements to make room) has different complexities depending on where the new element belongs. State the best case (insert at the very end, no shifting needed) and worst case (insert at the very beginning, shifting every existing element) in Θ-notation, and explain why they differ.

View solution

EXERCISE 2

Prove `f(n) = 2n + 7` is `Θ(n)` by finding valid constants for both the O(n) bound and the Ω(n) bound, showing the smallest n_0 for which the O(n) bound (with `c=3`) first holds.

View solution

EXERCISE 3

A colleague says "our hash table lookups are O(1), so performance is never a concern here." Using this chapter's own findings about best/worst/average case and the hash-collision example, explain what's missing from this claim.

📄 View solution

CHAPTER 4 QUICK REFERENCE

- **Big-Omega (Ω):** lower bound — `f(n) ≥ c·g(n)` for constants c, n_0
- **Big-Theta (Θ):** tight bound — both O and Ω hold simultaneously; the formal version of "report the tightest bound"
- O/Ω/Θ (bound type) and best/worst/average case (which input scenario) are **orthogonal** — never conflate them
- Linear search: Θ(1) best case, Θ(n) worst case, Θ(n) average case (even though average is only `(n+1)/2` comparisons — constants vanish in Big-notation)
- Worst-case bounds matter even when rare — they describe what an adversary or unlucky data can force, not just typical behavior (hash-collision DoS as a real example)
- **Next chapter:** Recursive algorithms and recurrence relations

CHAPTER 5 OF 10

Recursive Algorithms & Recurrence Relations

Algorithms & Complexity

Chapter 5 · Recursive Algorithms & Recurrence Relations

Chapters 3–4 analyzed code that repeats through loops. Recursive code repeats by calling itself — and analyzing its complexity needs a genuinely different tool: a **recurrence relation**, an equation defining a function's cost in terms of its own cost at smaller inputs.

From Code to Recurrence

THE GENERAL SHAPE

`T(n) = [number of recursive calls] × T([size of each subproblem]) + [non-recursive work per call]`, with a base case like `T(0) = O(1)` to anchor it.

Example 1: Linear Recursion

```
def countdown(n):
    if n <= 0:
        return
    print(n)
    countdown(n - 1)
```

One recursive call, on an input one smaller, plus O(1) work: `T(n) = T(n−1) + c`, `T(0) = c`.

Unrolling: `T(n) = T(n−1)+c = T(n−2)+2c = ... = T(0)+nc = c + nc = c(n+1)` — `O(n)`. Verified directly: `countdown(10)` makes exactly **11** calls (n=10 down through n=0) — matching `n+1`.

Proving the Solution — Exactly Discrete Mathematics Fundamentals Chapter 8's Own Technique

Claim: `T(n) = c(n+1)` for all `n ≥ 0`. This is proven by **induction** — the same base-case-plus-inductive-step structure that course used to prove a recursive function correct, now applied to prove a recursive function's *cost*:

STEP	WORKING
Base case	`T(0) = c(0+1) = c` — matches the definition exactly
Inductive hypothesis	Assume `T(k) = c(k+1)` for some `k ≥ 0`
Inductive step	`T(k+1) = T(k) + c = c(k+1) + c = c(k+2) = c((k+1)+1)` — matches the formula at `n=k+1`

Both steps hold, so `T(n) = c(n+1)` is proven for every `n` — not just checked on a few examples.

Example 2: Naive Fibonacci — Exponential Blowup

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Two recursive calls, each on a slightly smaller input, plus O(1) work: `T(n) = T(n-1) + T(n-2) + c`.

A QUICK, HONEST UPPER BOUND — NOT THE TIGHTEST ONE

Since `T(n-2) < T(n-1)`, replacing both terms with the larger one gives `T(n) ≤ 2T(n-1) + c`, which unrolls the same way as Example 1's pattern to `O(2ⁿ)`. This is a valid, easily-derived upper bound — the actual tight bound is `Θ(φⁿ)` where `φ ≈ 1.618` (the golden ratio), genuinely smaller than `2ⁿ` but harder to derive by hand. `O(2ⁿ)` is honest and sufficient for this course's own scope; it just isn't the tightest possible statement.

Real call counts confirm the exponential shape, even below the loose `2ⁿ` ceiling:

N	ACTUAL CALLS	2^n (UPPER BOUND)
10	177	1,024
15	1,973	32,768
20	21,891	1,048,576

Example 3: Divide-and-Conquer — A Forward Reference

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)  # O(n) to merge two sorted halves
```

T(N) = 2T(N/2) + O(N) — EXACTLY CHAPTER 6'S NEXT SUBJECT

Two recursive calls, each on *half* the input, plus O(n) work to merge the results back together. This shape — a fixed number of calls on a fraction of the input, plus polynomial extra work — is the single most common recurrence pattern in real divide-and-conquer algorithms, and it's exactly what Chapter 6's Master Theorem exists to solve directly, without unrolling by hand.

Recurrences in Code

```
def countdown_call_count(n):
    if n <= 0:
        return 1
    return 1 + countdown_call_count(n - 1)

print(countdown_call_count(10))  # 11 — matches T(n) = c(n+1) with c=1

def fib_call_count(n):
    if n <= 1:
        return 1
    return 1 + fib_call_count(n - 1) + fib_call_count(n - 2)

print(fib_call_count(10))   # 177 — well under 2^10 = 1024, but still exponential
```

Hands-On Exercises

EXERCISE 1

A recursive power function computes `xⁿ` as `x * power(x, n-1)`, with base case `power(x, 0) = 1`. Write its recurrence relation, then solve it by unrolling, following this chapter's own Example 1 method exactly.

View solution

EXERCISE 2

A recursive function makes 3 recursive calls, each on a third (`n/3`) of the original input, plus O(n) work to combine the results. Write this function's recurrence relation, following the exact shape of this chapter's own merge sort example.

View solution

EXERCISE 3

A recurrence is defined as `T(n) = T(n−1) + 2` , `T(0) = 5` . Using this chapter's own induction method, prove that `T(n) = 2n + 5` for all `n ≥ 0` , showing the base case and the full inductive step.

View solution

CHAPTER 5 QUICK REFERENCE

- **Recurrence relation:** `T(n) = [calls] × T([subproblem size]) + [non-recursive work]`, anchored by a base case
- **Linear recursion:** `T(n) = T(n−1) + c` unrolls to `O(n)` — provable exactly by Discrete Mathematics Fundamentals Chapter 8's own induction technique
- **Naive double recursion** (like Fibonacci): `T(n) = T(n−1) + T(n−2) + c` — a quick, honest upper bound is `O(2ⁿ)`, though the tight bound `Θ(ϕⁿ)` is smaller
- **Divide-and-conquer:** `T(n) = 2T(n/2) + O(n)` (merge sort's own shape) — Chapter 6's Master Theorem solves this pattern directly
- **Next chapter:** Solving recurrences — substitution and the Master Theorem

CHAPTER 6 OF 10

Solving Recurrences: Substitution & the Master Theorem

Algorithms & Complexity

Chapter 6 · Solving Recurrences: Substitution & the Master Theorem

Chapter 5 set up merge sort's own recurrence, `T(n) = 2T(n/2) + O(n)`, and left it unsolved. This chapter solves it two ways: the slow, honest way (full substitution), and the fast way every divide-and-conquer recurrence of this shape actually gets solved in practice — the Master Theorem.

Solving by Substitution — Merge Sort's Recurrence, Fully Unrolled

Starting from `T(n) = 2T(n/2) + cn`, repeatedly substituting the recurrence into itself:

STEP	EXPRESSION
1 level	`2T(n/2) + cn`
2 levels	`4T(n/4) + 2cn`
3 levels	`8T(n/8) + 3cn`
k levels	$2^kT(n/2^k) + k \cdot cn$

The pattern bottoms out when $n/2^k = 1$, i.e. $k = \log_2 n$. Substituting: $T(n) = 2^{(\log_2 n) \cdot T(1) + (\log_2 n) \cdot cn = n \cdot T(1) + cn \cdot \log_2 n$ — **Θ(n log n)**.

Verified directly (`c=1` , `T(1)=1`):

N	T(N)	$N + N \cdot \log_2 N$
4	12	12.0
8	32	32.0
16	80	80.0

The Master Theorem — A Fast-Path Formula

For any recurrence of the shape `T(n) = aT(n/b) + f(n)` , compare `f(n)` against `n^(log_b a)` :

CASE	CONDITION	RESULT
1	`f(n)` grows slower than `n^(log_b a)`	`T(n) = Θ(n^(log_b a))` — the recursion dominates
2	`f(n)` grows at the same rate as `n^(log_b a)`	`T(n) = Θ(n^(log_b a) · log n)` — perfectly balanced
3	`f(n)` grows faster than `n^(log_b a)`	`T(n) = Θ(f(n))` — the non-recursive work dominates

Applying It: Merge Sort, Confirmed Instantly

`T(n) = 2T(n/2) + cn`: `a=2`, `b=2`, `f(n) = Θ(n)`. `n^(log_b a) = n^(log₂2) = n¹ = n`. Since `f(n) = Θ(n)` matches `n^(log_b a) = n` exactly — **Case 2**: `T(n) = Θ(n · log n)`. Matches the full substitution above exactly, without needing to unroll anything.

Applying It: Binary Search, Confirmed Against Chapter 1

`T(n) = T(n/2) + O(1)`: `a=1`, `b=2`, `f(n) = Θ(1) = Θ(n⁰)`. `n^(log_b a) = n^(log₂1) = n⁰ = 1`. `f(n)` matches — **Case 2**: `T(n) = Θ(n⁰ · log n) = Θ(log n)`. Exactly Chapter 1's own opening figure, now derived formally rather than just quoted.

Applying It: A Genuine Case 1 — When Recursion Dominates

`T(n) = 4T(n/2) + n`: `a=4`, `b=2`, `f(n) = Θ(n)`. `n^(log_b a) = n^(log₂4) = n²`. Since `f(n) = n` grows strictly *slower* than `n²` — **Case 1**: `T(n) = Θ(n²)`, the recursive branching dominates entirely; the linear extra work barely matters.

> **VERIFIED BY WATCHING THE RATIO CONVERGE**
>
> Computing `T(n)` directly and dividing by `n²`: at `n=8`, ratio ≈ 1.875; at `n=32`, ≈ 1.969; at `n=64`, ≈ 1.984 — steadily converging toward a constant, exactly Chapter 2's own signature of a correct Θ classification, not drifting toward 0 or infinity.

Solving Chapter 5's Own Unsolved Exercise

Chapter 5's Exercise 2 asked only to *write* `T(n) = 3T(n/3) + O(n)`, promising the Master Theorem would solve it here. `a=3`, `b=3`, `f(n) = Θ(n)`. `n^(log_b a) = n^(log₃3) = n¹ = n`. `f(n)` matches — **Case 2**: `T(n) = Θ(n log n)`, the exact same class as merge sort, despite splitting into three pieces instead of two.

The Master Theorem in Code

```python
import math

def master_theorem_exponent(a, b):
    return math.log(a, b)   # n^(log_b a)

print(master_theorem_exponent(2, 2))  # 1.0 -- merge sort: n^1 = n
print(master_theorem_exponent(1, 2))  # 0.0 -- binary search: n^0 = 1
print(master_theorem_exponent(4, 2))  # 2.0 -- Case 1 example: n^2
print(master_theorem_exponent(3, 3))  # 1.0 -- Chapter 5's own exercise: n^1 = n
```

Hands-On Exercises

EXERCISE 1

For `T(n) = 4T(n/2) + n`, this chapter's own Case 1 example, verify by direct substitution/unrolling that `T(8) = 120` (with `T(1) = 1`), and confirm this is consistent with the Master Theorem's `Θ(n²)` prediction (compare `120` to `8² = 64` and note the constant-factor gap is expected).

View solution

EXERCISE 2

Apply the Master Theorem to `T(n) = T(n/2) + n²`. Compute `a`, `b`, `n^(log_b a)`, compare it to `f(n) = n²`, identify which case applies, and state the resulting Θ class.

View solution

EXERCISE 3

A recursive algorithm makes 8 recursive calls, each on a half-sized (`n/2`) subproblem, plus `O(n²)` non-recursive work. Write its recurrence, apply the Master Theorem, and state the resulting Θ class.

View solution

CHAPTER 6 QUICK REFERENCE

- **Substitution:** unroll `T(n) = 2T(n/2) + cn` level by level until it bottoms out at `k = log₂n` levels — gives `Θ(n log n)`
- **Master Theorem:** for `T(n) = aT(n/b) + f(n)`, compare `f(n)` to `n^(log_b a)` — three cases, whichever grows faster (or ties) wins
- Case 2 (tie) confirms both merge sort (`Θ(n log n)`) and binary search (`Θ(log n)`) without unrolling by hand
- Case 1 (recursion dominates) gives `Θ(n²)` for `T(n)=4T(n/2)+n`, verified by a ratio converging to a constant

- Chapter 5's own unresolved `3T(n/3)+O(n)` resolves to `0(n log n)` — Case 2 again, just with a different `a` and `b`
- **Next chapter:** Common complexity classes in practice — searching and sorting

CHAPTER 7 OF 10

Common Complexity Classes in Practice: Searching & Sorting

Algorithms & Complexity

Chapter 7 · Common Complexity Classes in Practice: Searching & Sorting

Binary search and sorting have been referenced throughout this course — Chapter 1's opening comparison, Chapter 3's loop gotchas, Chapter 5's recurrence setup, Chapter 6's Master Theorem proof. This chapter finally gives them their own full treatment, with real measured numbers instead of just formulas.

Binary Search — A Real Trace

Searching a sorted 15-element array (values 1–15) for the target `12`:

STEP	LO	HI	MID INDEX	VALUE AT MID	DECISION
1	0	14	7	8	8 < 12 → search right half
2	8	14	11	12	Found

Just **2** comparisons to find one element among 15 — each step throws away half the remaining candidates, exactly `O(log n)`, formally confirmed by the Master Theorem in Chapter 6.

> **THE PRECONDITION THAT MAKES IT ALL WORK**
>
> Binary search only works on **already-sorted** data — the halving logic depends entirely on knowing which half a target could be in. Sorting first costs something upfront (this chapter's own second half), but pays for itself the moment more than one search is needed against the same data.

Sorting: O(n^2) vs. O(n log n), Measured Directly

Bubble sort compares adjacent elements repeatedly — simple, but Chapter 3's own nested-loop pattern, giving `O(n²)`. Merge sort (Chapters 5–6) is `O(n log n)`. Rather than trust the formulas alone, here are actual comparison counts on random data:

N	BUBBLE SORT COMPARISONS	MERGE SORT COMPARISONS	RATIO
10	45	24	1.9×
100	4,950	534	9.3×
1,000	499,500	8,708	57.4×

THE GAP DOESN'T JUST GROW — IT GROWS FASTER THAN N ITSELF

Going from n=10 to n=1,000 (a 100× increase in input), the gap between the two algorithms widened from under 2× to over 57×. This is exactly what Chapter 2's own growth-rate table predicted in the abstract — now confirmed with real comparison counts on real data, not just formulas.

Quicksort: An Honest Average-vs-Worst-Case Gap

Quicksort is, in typical practice, one of the fastest sorting algorithms available — `O(n log n)` **average case**. But Chapter 4's own best/worst/average distinction applies directly here:

QUICKSORT'S WORST CASE IS GENUINELY $O(N^2)$

If the "pivot" element quicksort partitions around is consistently a poor choice — the smallest or largest remaining value, over and over — the algorithm degrades to $O(n^2)$, the exact same class as bubble sort. This happens, notably, on already-sorted or specifically-crafted adversarial input if the pivot selection is naive. Real implementations guard against this with randomized or median-of-three pivot selection specifically to make the worst case vanishingly unlikely in practice — but it remains a real, worth-knowing possibility, not a purely theoretical footnote.

How Low Can Sorting Go? The Comparison-Based Lower Bound

Merge sort's `O(n log n)` isn't just a good result — it's provably close to the best any comparison-based sort can ever achieve. There are `n!` possible orderings of `n` distinct elements (Discrete Mathematics Fundamentals' own permutations, Chapter 9 of that course), and each comparison can only rule out at most half the remaining possibilities. Distinguishing among `n!` orderings therefore needs at least $\log_2(n!)$ comparisons in the worst case.

$\log_2(N!)$ IS ITSELF $\Theta(N \log N)$

$\log_2(n!)$ and $n \log_2 n$ grow at the same rate — at n=1,000, $\log_2(1000!) \approx 8{,}529$ against $1000 \times \log_2 1000 \approx 9{,}966$, a ratio of about 0.86 and climbing toward a stable constant as n grows. This means **no comparison-based sorting algorithm can ever beat Θ(n log n) in the worst case** — merge sort isn't just a good algorithm, it's asymptotically optimal among comparison-based approaches.

Searching & Sorting in Code

```
def binary_search(arr, target):
    lo, hi = 0, len(arr) - 1
    steps = 0
    while lo <= hi:
        steps += 1
        mid = (lo + hi) // 2
        if arr[mid] == target:
            return mid, steps
        elif arr[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1, steps

arr = list(range(1, 16))
print(binary_search(arr, 12))  # (11, 2) — index 11, found in 2 steps

# Comparison-based lower bound, for any n
import math
def min_comparisons_needed(n):
    return math.log2(math.factorial(n))

print(min_comparisons_needed(1000))  # ~8529 -- close to 1000*log2(1000) ~ 9966
```

Hands-On Exercises

EXERCISE 1

Trace binary search on the sorted array `[3, 7, 11, 15, 19, 23, 27, 31]` (8 elements) searching for the target `23`. Show each step's `lo`, `hi`, `mid`, and decision, following this chapter's own trace format, and state the total number of comparisons needed.

View solution

EXERCISE 2

Using this chapter's own measured bubble-sort-vs-merge-sort table, estimate roughly how large the comparison-count gap would be at `n = 10,000` if the pattern of widening ratios continues (you don't need to run the code — reason from how the ratio grew between the three given data points). Explain your reasoning.

View solution

EXERCISE 3

A teammate says "quicksort is always faster than merge sort since it's the industry-standard fast sort." Using this chapter's own quicksort caveat and Chapter 4's own best/worst-case distinction, explain what's missing from this claim, and describe one concrete situation where quicksort's real-world performance could disappoint.

View solution

CHAPTER 7 QUICK REFERENCE

- **Binary search:** Θ(log n), requires sorted input — a 15-element array searched in just 2 comparisons
- **Bubble sort:** $O(n^2)$; **merge sort:** O(n log n) — measured directly, the gap widened from 1.9× to 57.4× going from n=10 to n=1,000
- **Quicksort:** O(n log n) average case, but a genuine $O(n^2)$ worst case with poor pivot choices — real implementations guard against this, but it's not purely theoretical
- **Comparison-based lower bound:** Ω(n log n) — no comparison sort can ever beat this, since distinguishing n! orderings needs at least $\log_2(n!)$ comparisons, and $\log_2(n!)$ is itself Θ(n log n)
- Merge sort is therefore asymptotically optimal among comparison-based sorts, not just "pretty good"
- **Next chapter:** Space complexity and amortized analysis

CHAPTER 8 OF 10

Space Complexity & Amortized Analysis

Algorithms & Complexity

Chapter 8 · Space Complexity & Amortized Analysis

Every chapter so far measured time. This chapter measures the other resource that matters just as much in practice — memory — and then covers a genuinely different kind of question: what happens when a single operation is occasionally expensive, but only rarely?

Auxiliary Space vs. Input Space

THE KEY DISTINCTION

Input space — the memory needed just to store the input — is usually not counted, since it's given regardless of the algorithm. **Auxiliary space** is the *extra* memory an algorithm allocates beyond that input. Space complexity almost always means auxiliary space.

Bubble sort (Chapter 7) sorts in place — a couple of temporary variables for swapping, regardless of array size: `O(1)` auxiliary space, even though it touches all `O(n)` of the input.

Merge Sort's Hidden Cost: O(n) Space

A REAL TIME-VS-SPACE TRADEOFF, NOT A FREE UPGRADE

Merge sort's merging step (Chapters 5–7) needs a temporary array to hold the merged result before copying it back — `O(n)` auxiliary space. Bubble sort's `O(n²)` time comes with only `O(1)` space; merge sort's much better `O(n log n)` time comes at the cost of genuine extra memory. Neither algorithm is simply "better" in every dimension — this is exactly the kind of tradeoff a real engineering decision has to weigh.

Recursive Call Stack Space

Every recursive call adds a frame to the call stack, and that frame occupies real memory until the call returns. Chapter 5's own countdown function makes `n` nested calls before hitting its base case — `O(n)` auxiliary space, purely from stacked call frames, even though it allocates nothing else.

A GENUINELY NON-OBVIOUS RESULT: NAIVE FIBONACCI USES ONLY O(N) SPACE

Chapter 5's naive Fibonacci makes an exponential *total* number of calls — but at any single moment, the call stack only holds calls along *one* path from the root down to a base case, since recursion explores depth-first: one branch fully unwinds before its sibling begins. Verified directly: `fib(20)` makes 21,891 total calls over its full execution, yet its call stack never exceeds depth **19**. Time (total work done) and space (maximum simultaneous memory) are genuinely different measurements — a function can be catastrophically slow while still being memory-cheap.

Time and Space Side by Side

ALGORITHM	TIME	SPACE	WHY
Bubble sort	$O(n^2)$	O(1)	Sorts in place, a few temp variables
Merge sort	O(n log n)	O(n)	Temporary arrays needed for merging
Countdown (Ch.5)	O(n)	O(n)	Call stack depth grows with n
Naive Fibonacci (Ch.5)	$O(2^n)$	O(n)	Exponential total calls, but only linear stack depth at once

Amortized Analysis: The Dynamic Array

A dynamic array (like Python's own list) grows by **doubling** its capacity whenever it fills up — copying every existing element into a new, larger array. That single resize is `O(n)`. But it doesn't happen on every append.

AMORTIZED ANALYSIS ASKS A DIFFERENT QUESTION

Not "what does the worst single operation cost," but "what's the *average* cost per operation across a long sequence of them" — even when some individual operations are genuinely expensive.

Simulating 16 appends, starting from capacity 1 and doubling each time it fills:

RESIZE EVENT	ELEMENTS COPIED
Capacity 1 → 2	1
Capacity 2 → 4	2
Capacity 4 → 8	4
Capacity 8 → 16	8

Total copy cost: `1+2+4+8 = 15`. Plus 16 individual element placements: `15 + 16 = 31` total operations for 16 appends — **≈1.94 operations per append**, even though the single most expensive append (the one triggering the capacity-8-to-16 resize) cost 9 operations on its own.

THE RATIO STAYS NEAR A SMALL CONSTANT AS N GROWS — O(1) AMORTIZED

At `n=100`: 227 total operations, ≈2.27 per append. At `n=1,000`: 2,023 total operations, ≈2.02 per append. The per-append cost doesn't grow with `n` — it hovers around 2, confirming `O(1)` **amortized** time, even though any single append can, rarely, cost `O(n)`.

Space & Amortized Analysis in Code

```
def simulate_dynamic_array(n):
    capacity, size, total_ops = 1, 0, 0
    for _ in range(n):
        if size == capacity:
            total_ops += capacity   # cost of copying on resize
            capacity *= 2
        total_ops += 1              # cost of placing the new element
        size += 1
    return total_ops

n = 1000
print(simulate_dynamic_array(n) / n)  # ~2.02 -- O(1) amortized

def fib_max_depth(n, depth=0, tracker=None):
    if tracker is None:
        tracker = [0]
    tracker[0] = max(tracker[0], depth)
    if n > 1:
        fib_max_depth(n - 1, depth + 1, tracker)
        fib_max_depth(n - 2, depth + 1, tracker)
    return tracker[0]

print(fib_max_depth(20))  # 19 -- linear space despite exponential total calls
```

Hands-On Exercises

EXERCISE 1

A function `reverse_new(arr)` builds and returns a brand-new list containing `arr`'s elements in reverse order. A second function `reverse_in_place(arr)` swaps elements within the same array using two index pointers moving toward each other. State the auxiliary space complexity of each, and explain the difference using this chapter's own input-space-vs-auxiliary-space distinction.

View solution

EXERCISE 2

Using this chapter's own dynamic-array-doubling method, compute the total operation count and the amortized cost per append for `n = 32` appends (starting from capacity 1, doubling at each resize). Show each resize event's own copy cost.

View solution

EXERCISE 3

Explain, in your own words, why naive recursive Fibonacci uses only O(n) auxiliary space despite making $O(2^n)$ total function calls over its full execution. Ground your answer in this chapter's own distinction between total work done (time) and maximum simultaneous memory in use (space).

View solution

CHAPTER 8 QUICK REFERENCE

- **Auxiliary space** (extra memory beyond the input) is what space complexity almost always measures — input space is usually not counted
- Recursive calls consume stack space proportional to **recursion depth**, not total calls made — naive Fibonacci is $O(2^n)$ time but only O(n) space
- Merge sort's O(n log n) time comes with a real O(n) space cost, versus bubble sort's O(1) space at $O(n^2)$ time — a genuine tradeoff, not a strict improvement
- **Amortized analysis:** the average cost per operation across a long sequence, even when individual operations vary wildly
- Dynamic array doubling: occasional O(n) resizes, but O(1) *amortized* time per append — verified directly (≈2 operations per append, staying constant as n grows)
- **Next chapter:** Beyond polynomial time — exponential growth and a taste of P vs. NP

CHAPTER 9 OF 10

Beyond Polynomial Time: Exponential Growth & a Taste of P vs. NP

Algorithms & Complexity

Chapter 9 · Beyond Polynomial Time: Exponential Growth & a Taste of P vs. NP

Chapter 1 showed $O(2^n)$ overtaking $O(n^2)$ by 32,000× at just `n=20`. This chapter puts real units on that divergence — actual seconds, days, years — and uses it to introduce the honest, practical boundary between problems that are efficiently solvable and problems nobody has ever found an efficient way to solve.

How Long, Really? Real Wall-Clock Estimates

Assuming a computer doing a generous 1 billion operations per second:

N	$O(2^n)$ OPERATIONS	ESTIMATED TIME
20	≈1.05 million	Under a millisecond
30	≈1.07 billion	≈1.1 seconds
50	≈1.13 quadrillion	≈13 days
100	≈1.27×10^{30}	≈2,911× the age of the universe

THIS ISN'T A HARDWARE PROBLEM

A computer a million times faster only pushes that `n=100` estimate down to "merely" about 3 million years — still catastrophically longer than any realistic patience. Exponential growth genuinely cannot be out-engineered with faster hardware; only a fundamentally different (polynomial) algorithm changes the picture at all, exactly Chapter 1's own "why choosing a better algorithm matters more than buying speed" point, now with real units attached.

Factorial growth is even more punishing:

N	$O(N!)$ OPERATIONS	ESTIMATED TIME
10	≈3.6 million	Under a millisecond
15	≈1.3 trillion	≈22 minutes
20	≈2.4×10^{18}	≈77 years

N	O(N!) OPERATIONS	ESTIMATED TIME
25	≈1.6×10^{25}	≈492 million years

A brute-force approach to something like the traveling salesman problem — checking every possible route ordering, an `O(n!)` approach — becomes physically hopeless before the input even reaches 25 cities.

A Taste of P vs. NP

TWO INFORMAL DEFINITIONS

P — problems *solvable* in polynomial time. Every algorithm this course has analyzed (searching, sorting, anything O(1) through O(n^k)) is in P.
NP — problems where a *proposed solution* can be *verified* in polynomial time, even if nobody knows how to efficiently find one.

Every problem in P is automatically in NP too — if you can solve something quickly, you can obviously check a proposed answer just as quickly (solve it yourself and compare). The genuinely open question, one of the most famous unsolved problems in mathematics and computer science: **is P = NP?** Is every problem whose solution can be quickly checked also quickly solvable — or are some problems truly, unavoidably harder to solve than to verify? Nobody has ever proven it either way; it's a Millennium Prize Problem with a $1 million reward still unclaimed.

TWO ACCESSIBLE NP EXAMPLES

Sudoku: given a completed grid, checking every row, column, and box is fast — but finding a solution from a mostly-empty puzzle can require real search. **Traveling salesman:** given a specific route, computing its total distance and checking it's under some target is fast — but finding the genuinely shortest route among `n!` possible orderings is exactly the factorial blowup shown above.

Certain problems — called **NP-complete** — are, in a precise sense, the "hardest" problems in NP: if *any* NP-complete problem were ever found to have a polynomial-time solution, that would prove `P = NP` and mean *every* problem in NP could suddenly be solved efficiently too. Traveling salesman is a classic NP-complete problem.

HONEST SCOPE NOTE

This is deliberately a light, informal introduction, not a real complexity theory course — proving a problem is NP-complete, formal reductions between problems, and the deep theory behind P vs. NP are genuinely out of scope here, exactly per Chapter 1's own boundary. The goal is awareness of the boundary and roughly what it means, not the machinery to navigate it formally.

The Practical Takeaway

WHY THIS MATTERS FOR REAL ENGINEERING DECISIONS

If a problem is known (or strongly suspected) to be NP-complete, the practical response isn't to keep hunting for a clever fast exact algorithm — decades of effort by the entire field haven't found one for any NP-complete problem. The real options are: solve it exactly only for genuinely small `n` (per this chapter's own tables, that means staying well under 20–30 for factorial-shaped problems), use an **approximation algorithm** that finds a provably good-enough answer quickly, or use a heuristic that works well in practice without any formal guarantee. Recognizing "this looks NP-complete" is itself a useful, actionable signal.

Real Numbers in Code

```
import math

OPS_PER_SECOND = 1e9

def estimate_seconds(operation_count):
    return operation_count / OPS_PER_SECOND

n = 100
seconds = estimate_seconds(2 ** n)
years = seconds / (365.25 * 24 * 3600)
print(f"{years:.3e} years")  # ~4.02e13 years

n2 = 20
seconds2 = estimate_seconds(math.factorial(n2))
print(f"{seconds2 / (365.25*24*3600):.1f} years")  # ~77.1 years
```

Hands-On Exercises

EXERCISE 1

Using this chapter's own 1-billion-operations-per-second assumption, estimate the running time of an $O(2^n)$ algorithm at `n = 40`. Express the result in a human-readable unit (seconds, minutes, hours, etc.), showing your calculation.

View solution

EXERCISE 2

Using the same assumption, estimate the running time of an O(n!) algorithm at `n = 15`, expressed in a human-readable unit. Compare it to this chapter's own $O(2^n)$ estimate at a similar-magnitude operation count, and comment on which grows faster for a given increase in n.

📄 View solution

EXERCISE 3

Using a Sudoku puzzle as a concrete example, explain in your own words the difference between "solving" a problem and "verifying" a proposed solution to it, and why this distinction is exactly what defines the class NP.

📄 View solution

CHAPTER 9 QUICK REFERENCE

- At 1 billion ops/sec, $O(2^n)$ reaches **2,911× the age of the universe** by `n=100` — no realistic hardware speedup rescues this
- O(n!) is even more punishing — 20! alone takes ≈77 years at the same rate
- **P:** solvable in polynomial time. **NP:** a proposed solution verifiable in polynomial time. Every P problem is in NP; whether the reverse holds (P = NP) is a famous unsolved problem
- **NP-complete** problems are the "hardest" in NP — solving any one in polynomial time would solve all of NP efficiently
- Sudoku and the traveling salesman problem are accessible, concrete NP examples — easy to verify, hard (as far as anyone knows) to solve
- **Practical takeaway:** a suspected NP-complete problem calls for approximation or heuristics, not an endless search for a fast exact algorithm
- **Next chapter:** Capstone — analyzing and comparing real algorithms

CHAPTER 10 OF 10

Capstone — Analyzing and Comparing Real Algorithms

Algorithms & Complexity

Chapter 10 · Capstone — Analyzing and Comparing Real Algorithms

One continuous worked problem, touching every chapter of this course in the order a real engineer would actually reach for each idea: finding duplicate values in a list of `n` user IDs, solved three genuinely different ways.

A Full Worked Comparison — Finding Duplicate User IDs

1 — FRAMING THE PROBLEM AND A FIRST GUESS (CH.2)

The obvious first approach: check every pair of elements for a match. Before writing any code, Chapter 2's own dominant-term intuition already previews the shape — comparing every element against every other element is going to land in the `O(n²)` family, the same shape as any nested "compare all pairs" structure.

2 — COUNTING THE BRUTE-FORCE OPERATIONS PRECISELY (CH.3)

Avoiding double-counting each pair, the brute-force approach compares element `i` against every element `j > i` — exactly Chapter 3's own triangular loop pattern. Total comparisons: `n(n-1)/2`. On a real test list of `n = 1,000` IDs (with 50 genuine duplicates planted in): **499,500** comparisons — matching `1000×999/2` exactly.

3 — BEST AND WORST CASE FOR BRUTE FORCE (CH.4)

Best case: the very first two elements checked happen to be duplicates — found in a single comparison, `Θ(1)`. Worst case: no duplicates exist at all, forcing every one of the `n(n-1)/2` comparisons to run to completion — `Θ(n²)`. The 499,500-comparison run above *is* the worst case, since it never gets to skip ahead once a match streak is found.

4 — A RECURSIVE ALTERNATIVE, AND ITS RECURRENCE (CH.5)

A divide-and-conquer idea: split the list in half, recursively find duplicates within each half, then separately handle duplicates that span across the two halves. In the shape Chapter 5 established: `T(n) = 2T(n/2) + [cost of the cross-half check]` — everything hinges on how cheaply that cross-half step can be done.

5 — MAKING THE CROSS-HALF CHECK CHEAP: SORT FIRST (CH.6)

If each half is sorted before combining — exactly merge sort's own merge step — checking for duplicates spanning the boundary becomes a single `O(n)` linear pass. That gives `T(n) = 2T(n/2) + O(n)` — Chapter 6's own Master Theorem Case 2, resolved instantly: `Θ(n log n)`, with no new derivation needed.

6 — FORMALIZING IT: SORT, THEN SCAN (CH.7)

This resolves into a clean two-step algorithm: sort the list (`O(n log n)`, using merge sort specifically — Chapter 7's own measured comparison counts showed it decisively beating bubble sort's `O(n²)` at any real scale), then a single linear scan checking each element against its neighbor for a match. On the same 1,000-ID test list: the scan itself takes just **999** comparisons, plus roughly **9,966** for the sort — both dramatically below brute force's 499,500.

7 — A THIRD APPROACH, AND A GENUINE SPACE TRADEOFF (CH.8)

A hash set offers a third option entirely: scan the list once, checking whether each element is already in a hash set before adding it — average-case `O(1)` per check (Chapter 4's own hash table discussion), giving `O(n)` total time. On the same test list: exactly **1,000** operations, one per element — even faster than sort-then-scan. But per Chapter 8's own auxiliary-space accounting, this approach needs a full `O(n)` hash set alongside the original list — a real memory cost the brute-force approach (needing only `O(1)` extra space) never pays.

8 — CAN ANYTHING BEAT O(N)? AN INHERENT LOWER BOUND (CH.9)

No — and this is provable without any deep complexity theory. Any correct algorithm for this problem must examine every element at least once (an unexamined element could always be the one hiding a duplicate), so `Ω(n)` is an unavoidable floor for *any* approach. The hash-set method's `O(n)` is therefore asymptotically optimal — a genuinely different, tighter lower bound than Chapter 7's own `Ω(n log n)` for comparison-based sorting, since this problem doesn't require comparison-based sorting at all. And

unlike Chapter 9's own traveling-salesman example, this problem sits comfortably in **P** — efficiently solvable, no NP-completeness in sight.

The Full Comparison, Side by Side

APPROACH	TIME	SPACE	MEASURED OPS (N=1,000)
Brute force	$\Theta(n^2)$	O(1)	499,500
Sort then scan	$\Theta(n \log n)$	O(n)*	≈10,965
Hash set	O(n) average	O(n)	1,000

**Using merge sort; an in-place O(n log n) sort would bring this down to O(1), a further tradeoff of its own.*

Brute force to hash set: a **499.5×** reduction in measured operations, for the exact same correct result — every chapter of this course, pointed at one real problem.

What This Course Doesn't Cover

In the interest of an honest accounting: a full catalog of every named algorithm and data structure, graph algorithms (reserved for this subject's own future Graph Theory course), and deep computational complexity theory beyond Chapter 9's light introduction were all named in Chapter 1 as deliberately out of scope. This course built the mathematical toolkit for *analyzing* any algorithm's efficiency, not an exhaustive tour of algorithms themselves.

This Course's Throughline, Restated

> **PREDICTING COST BEFORE RUNNING THE CODE**
>
> Every chapter in this course answered a version of the same question: as input grows, how does the required work actually grow — and can that be predicted directly from the code, before ever running it at scale? The capstone above used exactly eight ideas — growth rates, loop counting, best/worst case, recurrences, the Master Theorem, searching/sorting, space tradeoffs, and inherent lower bounds — to turn one ordinary-sounding problem into a genuine, quantified 499.5× improvement.

Where This Course Connects

This course is the direct mathematical foundation under every other course on this site that writes real code — Technical Support's own `perfdiag1` diagnoses slowness after the fact; this course predicts it beforehand. Within this subject's own future courses, Graph Theory will build directly on this course's recursion and Master Theorem material for traversal and pathfinding algorithms, and Number Theory & Cryptographic Math will lean on this course's own complexity-class vocabulary to explain why certain cryptographic problems are chosen specifically for their computational hardness.

Hands-On Exercises

EXERCISE 1

For a list of `n = 200` elements with no duplicates at all, compute the exact number of brute-force comparisons (Step 2's own formula) and the exact number of hash-set operations (Step 7's own approach). Compute the ratio between them.

View solution

EXERCISE 2

A teammate proposes a fourth approach: for each element, use binary search (Chapter 7) to check whether it already exists in a separately maintained sorted list, inserting it if not. Using this chapter's own space/time framework, analyze this approach's time complexity (consider the cost of both the binary search itself and of inserting into a sorted list — Chapter 4's own sorted-array insertion exercise is directly relevant) and compare it honestly to the hash-set approach.

View solution

EXERCISE 3

For each of the eight steps in this chapter's own worked comparison, name the specific topic it relied on, without looking back at the step labels — just from the description of what each step actually does.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked comparison:** growth-rate intuition (Ch.2) → precise loop counting (Ch.3) → best/worst case (Ch.4) → a recursive alternative (Ch.5) → the Master Theorem (Ch.6) → sort-then-scan formalized (Ch.7) → a hash-set tradeoff (Ch.8) → an inherent lower bound (Ch.9)
- Brute force ($\Theta(n^2)$) → sort-then-scan ($\Theta(n \log n)$) → hash set ($O(n)$ average) — a measured 499.5× reduction in operations at n=1,000, all solving the identical problem correctly

- Time and space are genuinely separate axes — the fastest approach (hash set) isn't automatically the cheapest in memory
- Different problems have different inherent lower bounds — Ω(n) here, versus Ω(n log n) for comparison-based sorting — neither is a universal floor
- Out of scope: a full algorithms catalog, graph algorithms (a future course), and deep complexity theory beyond Chapter 9's light introduction
- **Course complete** — Algorithms & Complexity, 10 chapters, from Big-O to P vs. NP