COMSC 2043: Induction and Recursion
Teacher Solution Guide

Jeremy Evert
Southwestern Oklahoma State University

October 28, 2025

Contents

Chapter 1

Foundations of Induction and Recursion — Teacher's Guide

1.1 Overview for Instructors

This chapter anchors students in the duality of **induction and recursion**. Your goal is to help them see that these two concepts are not separate tools, but complementary ways of describing repetition and self-reference.

A strong introduction here sets the tone for the rest of the course: students must leave with intuition, not just definitions. Encourage them to see recursion as "definition by self-reference" and induction as "proof by dominoes."

1.2 Teaching Objectives

By the end of this chapter, students should be able to:

- Explain the relationship between recursion and induction.
- Write a simple recursive definition (e.g., factorial).
- Construct and justify a basic proof by mathematical induction.
- Recognize how recursive algorithms reflect inductive proofs.

1.3 Section 1.1 — The Big Picture

Student Goal: Understand that recursion defines a process and induction proves it correct.

Teaching Note: Open class with a visual metaphor — line up real dominoes or use a slide animation. Show one falling into the next. Then write on the board:

> *Induction: proving all dominoes fall.*
> *Recursion: building the dominoes themselves.*

Encourage students to explain how one supports the other. For programming-minded learners, connect the idea to a recursive call stack: each call relies on the truth of smaller subproblems.

1.4 Section 1.2 — Key Ideas from Rosen's Chapter 5

Basis Step: Verify that $P(0)$ or $P(1)$ is true. This builds confidence in the foundation.

Inductive Step: Assume $P(k)$ and prove $P(k+1)$. This forms the "engine" of reasoning.

Strong Induction: Stress that this is not stronger logic, but broader assumption.

Recursive Definition: Let the process mirror induction — base case + recursive rule.

Structural Induction: For trees and grammars, emphasize that the same logic applies to structure.

Instructor Tip: Rosen's section on structural induction (Chapter 5.3) pairs beautifully with programming examples. Ask students how a parse tree or HTML document could be proven valid using the same logic.

1.5 Section 1.3 — Why This Matters

Students often treat induction as abstract until it's made concrete. Use examples that connect mathematical induction to computer science:

- Recursive definitions in Python or Java mirror inductive reasoning.
- Correctness proofs for loops and algorithms rely on inductive invariants.
- Sorting, searching, and even AI search trees can be analyzed inductively.

Misconception Watch: Students may think induction proves "by example." Clarify: *One base case and one domino rule are enough for infinitely many cases.*

1.6 Section 1.4 — Example: The Factorial Function

Recursive Definition:

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n > 0. \end{cases}$$

Proof by Induction: Show that $n! \geq 2^{n-1}$ for $n \geq 1$.

Base case: $1! = 1 \geq 2^0 = 1$ ✓

Inductive step: Assume $k! \geq 2^{k-1}$. Then $(k+1)! = (k+1)k! \geq (k+1)2^{k-1} \geq 2^k$.

Thus the property holds for all $n \geq 1$.

Instructor Strategy: 1. Work through this on the board line by line. 2. Have students complete the inequality on their own for $(k + 2)!$ to test comprehension. 3. Discuss what would break if the base case were omitted.

1.7 Section 1.5 — The Student Challenge

Challenge statement (student version):

> "Induction is not a leap of faith—it's a method of climbing an infinite ladder, one rung at a time."

Teacher Expansion: Encourage students to:

1. Write a recursive Python function for $n!$.
2. Prove its correctness using induction.
3. Identify the correspondence between code and proof:
   - Base case $\leftrightarrow$ if-statement for $n = 0$
   - Inductive step $\leftrightarrow$ recursive call to smaller n

Sample Code:

Listing 1.1: Recursive factorial function in Python

```python
def factorial(n):
    """Return n! recursively."""
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Ask students to trace 'factorial(4)' and list every call on the board. Then show how the recursion tree mirrors the structure of the inductive proof.

Extension: Have advanced students formalize the induction as a theorem:

$$\forall n \geq 0, \ \text{factorial}(n) = n!$$

1.8 Section 1.6 — Checkpoint Questions with Model Answers

1. **What are the two main steps of a proof by induction?**
   Answer: The basis step (prove the first case) and the inductive step (assume $P(k)$, then prove $P(k + 1)$).

2. **How is recursion related to induction?**
   Answer: Recursion defines a process in terms of smaller instances; induction proves that the process works for all instances.

3. **Give a real-world example of a recursive process.**
   Possible answers: Folding paper, Russian nesting dolls, family trees, the Tower of Hanoi, or fractal growth in nature.

4. **Can every recursive definition be proven correct using induction?**
   Answer: Yes—provided it terminates and is well-founded. Induction is the formal method used to prove the correctness of recursive definitions.

1.9 Teaching Discussion Prompts

- "Where do we see self-reference in the real world?"
- "If recursion builds, what does induction guarantee?"
- "What happens if a recursive function lacks a base case?"
- "How would you prove that your recursive function always terminates?"

1.10 Common Misconceptions

- **"Induction means guessing and checking."** $\rightarrow$ Clarify that it's logical deduction, not pattern recognition.
- **"Recursion runs forever."** $\rightarrow$ Only without a base case! Stress convergence and termination.
- **"Strong induction is different math."** $\rightarrow$ Emphasize it's the same principle with an expanded hypothesis.

1.11 Classroom Activity Ideas

1. Use Jupyter or Python to demo factorial recursion visually.
2. Have students form a "human call stack" — each student represents a recursive call.
3. Challenge groups to come up with non-math examples of recursion.
4. Close class by proving a fun pattern like $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ inductively.

1.12 Instructor Reflection

Induction is belief with a proof.

Reflect on how you framed induction not as a dry method but as a way of thinking. Students who "get it" here will see recursion everywhere—from fractals to AI to data structures.

Next Chapter: Recursive Algorithms — Turning Thought into Code

Chapter 2

Recursive Algorithms — Teacher's Commentary

2.1 Overview and Pedagogical Goals

This chapter invites students to think recursively, not just to code recursively. Our primary teaching objective is to help them *see* recursion as a pattern of thought: a problem divided into smaller, self-similar pieces that collectively build a solution. We emphasize three ingredients:

1. a clear **base case**, which guarantees termination,
2. a **recursive step** that simplifies the problem,
3. and a sense of **trust** in the recursion's correctness, usually grounded in induction.

When students conflate recursion with loops, it helps to remind them that recursion *is not just repetition* — it's **definition by self-reference**.

2.2 Discussion of Key Examples

2.2.1 Summation

The student text defines

$$S(n) = \begin{cases} 0, & n = 0, \\ n + S(n-1), & n > 0. \end{cases}$$

and the Python version mirrors this logic:

```python
def sum_to_n(n):
    if n == 0:
        return 0
    else:
        return n + sum_to_n(n - 1)
```

Teacher Notes. Encourage students to mentally trace $S(4)$:

$$S(4) = 4 + S(3) = 4 + 3 + S(2) = 4 + 3 + 2 + S(1) = 4 + 3 + 2 + 1 + S(0).$$

Highlight the idea of a *call stack*. Each call waits for its child to finish. This visualization builds intuition for later discussions of stack depth and resource use.

A useful classroom exercise is to draw this as a tree or stack diagram on the board, then trace its unwinding phase as results return upward.

2.2.2 Factorial

Students are already familiar with factorial from Chapter 1's induction proof, so this is a good place to reinforce the bridge between **inductive reasoning** and **recursive computation**. Both use a base case and an inductive/recursive step.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Common Student Pitfalls.

- Forgetting to include a base case, which causes infinite recursion.
- Using subtraction in the wrong direction, e.g., `factorial(n + 1)`.
- Confusing where the multiplication occurs (in the recursion or after).

Demonstrate visually: show how each call multiplies by n as the stack unwinds.

2.2.3 Fibonacci

Here, beauty meets cost. The simple recursive definition:

$$F(n) = \begin{cases} 0, & n = 0, \\ 1, & n = 1, \\ F(n-1) + F(n-2), & n > 1 \end{cases}$$

translates directly into Python:

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

Teacher Notes. Use this example to illustrate the exponential explosion of calls. Ask the class: "How many calls occur when computing `fib(5)`?" Let them discover the repeating subcalls: `fib(3)` appears multiple times. This repetition sets up the motivation for **memoization** in later chapters.

Extension Activity. Have students modify the function to count calls:

```
count = 0
def fib_count(n):
    global count
    count += 1
    if n <= 1:
        return n
    return fib_count(n - 1) + fib_count(n - 2)
```

Then run `fib`$_count(10) and discuss how fast the count grows.$

2.3 Tracing Recursion and Cost Analysis

Tracing Exercise. Trace $S(4)$ and $fib(5)$ by hand or with Python's call visualization tools (e.g., pycallgraph or a simple print statement).

Analyzing Recursive Cost. The factorial's recurrence $T(n) = T(n-1) + O(1)$ resolves to $O(n)$, while Fibonacci's $T(n) = T(n-1) + T(n-2) + O(1)$ grows as $O(2^n)$.

Ask students to identify where each new call is spawned and how much redundant work occurs.

2.4 Challenge Problem Solutions and Commentary

1. Sum of digits.

   ```
   def sum_digits(n):
       if n < 10:
           return n
       else:
           return n % 10 + sum_digits(n // 10)
   ```

   This teaches recursive decomposition on numeric structures.

2. Memoized Fibonacci.

   ```
   def fib_memo(n, memo=None):
       if memo is None:
           memo = {}
       if n in memo:
   ```

```
            return memo[n]
        if n <= 1:
            memo[n] = n
            return n
        memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
        return memo[n]
```

Explain how the recursion tree collapses into a line|each distinct input is computed once.

3. Trace fib(5). The recursion tree has 15 calls in total. Have students label each call and result. Visualizing overlapping subproblems reinforces why memoization works.

4. Prove $O(n)$ runtime of memoized Fibonacci. Each integer from 0 to n is evaluated once, with $O(1)$ work per call. Therefore, $T(n) = O(n)$.

2.5 Teaching Reflections

Students often oscillate between fascination and frustration with recursion. Invite them to use analogies:

- Recursion as storytelling|each call writes a paragraph, then returns to complete the chapter.
- The call stack as memory of unfinished business.
- Induction as the \legal proof" that justifies recursion's correctness.

Ending thought: once students grasp recursion, they begin to think like the computer| and like a mathematician.

Chapter 3

The Fibonacci Sequence – Nature's Algorithm — Teacher's Commentary

3.1 The Story Beneath the Spiral

The Fibonacci sequence is where mathematics stops being a cold ledger and begins to hum. It is the pattern that flowers whisper and pinecones carve; it is the mathematical heartbeat of reproduction, rhythm, and recursion.

Begin your lesson by asking: *What does it mean for something to build itself from itself?* Let students suggest everyday examples|stories that fold back, family trees, mirrors, Russian nesting dolls. This primes the recursive intuition before a single formula is written.

> Pedagogical hint: Treat Fibonacci not as a new sequence, but as the first time the students meet a truly self-referential idea in code.

3.2 Seeing the Pattern Emerge

Start with the simple rule:

$$F(n) = F(n-1) + F(n-2), \quad F(0) = 0,\ F(1) = 1.$$

Ask them to compute $F(2)$, $F(3)$, $F(4)$ by hand on the board. Then quietly step aside and let the pattern take over. The delight comes from watching the class realize they can keep going forever, but also that every new term depends on the last two|the past always haunts the future.

> Common misconception: Students often believe that each term is computed once. In the recursive version, it isn't! Highlight that every call to fib(n) may re-summon its siblings many times.

3.3 A Gentle Python Beginning

Transition into code gently. Begin with the most human expression of recursion|a definition that reads like English.

Listing 3.1: Naive Fibonacci definition

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

Invite them to trace fib(5). Encourage sketching a recursion tree on the whiteboard. Each branch is a call; each leaf a base case. This is where students begin to *see* recursion rather than merely compute it.

> Teaching tip: Ask the class: \If we could hear the computer think, what would this sound like?" The answer: a chorus of overlapping echoes.

3.4 Counting the Chaos

Introduce a counting function that measures how many recursive calls occur.

Listing 3.2: Instrumented Fibonacci counter

```
calls = 0

def fib_count(n):
    global calls
    calls += 1
    if n <= 1:
        return n
    else:
        return fib_count(n-1) + fib_count(n-2)

calls = 0
fib_count(10)
print("Total␣calls:", calls)
```

Have students predict before running. Most guess \maybe 10 or 20." The real answer|177|usually triggers laughter and disbelief. This is where Big-O starts to feel real.

> Discussion prompt: How does this explosion relate to the branches of the recursion tree? Which values are recomputed? How could we prevent that?

3.5 The Moment of Memoization

Before revealing the fix, invite a student to guess how the computer might \remember" prior results. Then, introduce memoization as a form of *memory with manners*|a polite way for a function to say, \Oh wait, I've done that already."

Listing 3.3: Memoized Fibonacci

```
memo = {0: 0, 1: 1}

def fib_memo(n):
    if n not in memo:
        memo[n] = fib_memo(n-1) + fib_memo(n-2)
    return memo[n]
```

Ask the class to predict how many calls this version makes for n=30. Then run it|only 59! The algorithm now feels like a wise old sage: still recursive, but reflective.

3.6 Reflection and Connection

Bring the conversation back to nature: how leaves, shells, and pinecones follow Fibonacci because growth builds upon growth. The algorithm's *mathematical economy* mirrors biological efficiency.

> Teaching insight: Recursion is not just a technique|it's a philosophy. Every complex system that repeats patterns of its past to create its future is recursive in spirit.

3.7 Bridging to the Next Chapter

Close by preparing them for Chapter ??, where they will measure recursion's cost.
Give them a final experiment:

- Run both fib(30) and fib_memo(30).
- Compare the time difference.
- Discuss which grows faster and why.

Encourage curiosity: \What if we plotted the number of calls vs. n? What shape would it form?" (Answer: an exponential mountain for naive recursion, a gentle hill for memoization.)

3.8 Teacher's Reflection

Mantra: Teach recursion as poetry first, algorithm later.

Students remember stories longer than syntax. Fibonacci is your gateway to the emotional side of algorithms|a reminder that patterns can be both efficient and beautiful.

End this lesson with the story of rabbits, or with the sunflower's spiral. Smile, because the math itself is smiling back.

Chapter 4

Measuring the Cost of Recursion — Teacher's Commentary

4.1 Teaching Overview

This chapter moves from wonder to measurement. Students have seen recursion bloom; now they must quantify its cost. As an instructor, your role is to turn curiosity into data.

Key goal: help students feel the *weight* of recursion. The Tracker class turns invisible stack frames into visible numbers.

4.2 Pedagogical Setup

Before class:

- Have students run the code measure_fibonacci() from Chapter 4 of the student workbook.
- Ask them to predict: which will grow faster, calls or additions? Which chart will be exponential?
- Display the recursive vs. iterative plots side by side.

Teaching tip: Pause at each plot. Ask: *What does the shape tell you? What story does this data whisper?*

4.3 Instructor Notes on Code

The DataTracker class is not about performance optimization; it's about cognitive visibility. Encourage students to:

- Trace which lines increment counters.
- Modify it to count multiplications or return depths.

- Compare run-to-run variation.

Then connect the dots to Big-O notation | it's not abstract now; it's empirical.

4.4 Common Pitfalls

- Students forget to reset counters between runs.
- They compare recursive vs iterative without realizing base cases differ.
- They confuse stack growth with data growth | emphasize call count vs. memory use.

4.5 Extension Ideas

- Challenge them to add memoization and measure again.
- Introduce timing for each n and graph log-scale axes to reveal asymptotic behavior.
- Ask: \How could we verify this with induction?"

4.6 Reflection Prompts

> Recursion is the art of calling yourself until you learn something new each time.

- What does the graph of recursion *feel* like?
- When does elegance become inefficiency?
- Can you love a slow algorithm if it teaches you something fast?

4.7 Instructor Reflection

This chapter transforms recursion from poetry into physics. By measuring time, calls, and assignments, you bridge emotion and evidence.

Encourage students to end the session by answering one final question: *If your code were a living thing, what would it remember between calls?*

Chapter 5

Understanding Big-O Through Fibonacci — Teacher's Commentary

5.1 Overview and Teaching Arc

This chapter is where students meet efficiency as a *character* in the recursive story. They have already seen recursion as poetry and measurement as physics; now, they will see *Big-O* as philosophy---a way to describe how beauty behaves when stretched toward infinity.

Your task as instructor: guide them from the emotional hum of Fibonacci's pattern into the cool, analytic light of asymptotic reasoning.

5.2 Connecting Concept to Code

Students now have empirical data from the Tracker experiment. Invite them to revisit their CSV or plotted graphs.

- Ask: ''What shape does recursion's time take?''
- Ask: ''When does wonder become waste?''

Explain that Big-O abstracts the details but keeps the melody. Every measurement they made is one note in the infinite symphony of growth rates.

5.3 Discussion of Key Results

Show the two cost equations on the board:

$$T_{\texttt{naive}}(n) = T(n-1) + T(n-2) + O(1)$$
$$T_{\texttt{memoized}}(n) = T(n-1) + O(1)$$

Then draw the moral: recursion without memory grows like rumor; recursion with memory grows like wisdom.

Encourage them to sketch the exponential and linear curves on the same axes. When n is small, both seem gentle; by $n = 30$, the exponential explodes off the chart. Let that contrast speak louder than any proof.

5.4 Teaching Activities

Classroom Experiment

Run all three implementations side by side: naive, memoized, and iterative. Project a live graph if possible.

1. Ask students to predict the runtime for $n = 35$.
2. Let them timeit the functions.
3. Pause for that stunned silence when the recursive one stalls.

Whiteboard Derivation

Derive the recurrence together. Each branch in the Fibonacci tree births two more until leaves overrun the forest. Then prune with memoization and show how the forest becomes a single vine. This visualization ties Big-O directly to recursive structure.

5.5 Analogies and Metaphors

> Big-O is the mathematician's weather report: it doesn't tell you whether it will rain tomorrow, but it tells you how storms grow.

Offer students tangible comparisons:

- $O(n)$ --- a calm climb up a hill.
- $O(n^2)$ --- a staircase that doubles back.
- $O(2^n)$ --- a volcano; breathtaking, but deadly for laptops.

Encourage them to describe algorithmic growth using their own metaphors. Let them own the language.

5.6 Extension and Reflection

Push beyond the graph:

- What other algorithms behave exponentially?

- Can memoization be seen as evolution|memory as survival strategy?
- How does empirical timing confirm (or challenge) theoretical Big-O?

End with a reflective prompt:

> If recursion is a story that repeats itself, Big-O is the measure of how long the story can keep being told.

5.7 Instructor Notes

- Run the full experiment early in class; discuss results mid-lesson.
- Encourage students to annotate their plots with asymptotic labels.
- Reinforce that Big-O compares shapes, not seconds.

5.8 Transition to Chapter 6

The next chapter turns the analysis inward: *Memoization as Memory of the Mind.* Students will see that remembering one's past|both in code and in cognition| transforms impossible problems into elegant ones.

Chapter 6

ch06 memoization mind solution

This is a placeholder for ch06$_{m}emoization_{m}ind_{s}olution.tex$.

Chapter 7

ch07 big o meets reality solution

This is a placeholder for ch07$_{b}ig_{om}eets_{r}eality_{s}olution.tex$.

Chapter 8

ch08 teaching tips and assessments

This is a placeholder for ch08$_{t}eaching_{t}ips_{a}nd_{a}ssessments.tex$.